Enabling Scalable Online User Interaction through Data Warehousing of Interaction Histories

A Thesis
Presented to
The Academic Faculty

by

Helen Thomas

In Partial Fulfillment
of the Requirements for the Degree of
Doctor of Philosophy in Management

Georgia Institute of Technology
November 2001

Enabling Scalable Online User Interaction through Data Warehousing of Interaction Histories

Approved:

[illegible]

Anindya Datta, Chairman

[illegible]

Sri Narasimhan

[illegible]

Saby Mitra

[illegible]

Samit Soni

[illegible]

Sham Navathe

Date Approved 11-27-2001

Contents

List of Tables

List of Figures

Preface

Online user interaction is a topic of considerable current interest, both from a research as well as from a practical perspective. Virtually all online user interaction technologies in use today (e.g., personalization and customer relationship management software) are based on the notion of storing as much historical customer session data as possible, and then querying this data store in order to react to customers (e.g., offering a discount on an item that the user has shown interest in). The holy grail of online user interaction is an environment where fine-grained, detailed historical session data can be queried based on current online behavioral patterns for use in formulating near real-time responses. Unfortunately, most existing online user interaction technologies are unable to scale to support the high user loads and large volumes of customer data that are typical of many e-commerce sites today. Providing true online user interaction requires that data be retrieved from large persistent databases within subsecond time frames, and typically this must be done under heavy user loads. Thus, the primary bottleneck lies in existing database systems that cannot effectively support these requirements.

This research attempts to present an approach to perform *true* online user interaction. The proposed framework entails: 1) observing specific instances of online behavior, 2) correlating this specific behavior with the vast amounts of historical behavior collected over time, and 3) reacting to the user. Our solution approach consists of two key ideas: 1) a data warehouse to store historical behavior, and 2) rule caching to track online behavior and correlate with the historical data.

The data warehouse is a critical component in such an online user interaction system. The warehouse must store large volumes of data and provide fast response times for queries on this data. The warehouse is typically vast (e.g., several hundred gigabytes to a few terabytes in size), and the query operations required on this data will typically be complex, usually requiring joins of several large tables. In addition, given the interactive nature of the Web, query results must be delivered within sub-second time frames, even in the presence of heavy user loads (e.g., thousands to tens of thousands of simultaneous users). We address these issues by presenting a new data storage and retrieval paradigm for data warehouses, referred to as *DataIndexes*. In addition to the DataIndex structures, we present a set of efficient algorithms for performing join queries using the DataIndex structures. A mathematical analysis is presented which categorizes the cost of query evaluation for common classes of queries using DataIndexes. In addition, the results of an implementation based on DataIndexes is presented. Both the analysis and the implementation results indicate that DataIndexes significantly outperform existing indexing structures in many cases, and that they can provide the subsecond response times required for online user interaction.

While the data warehouse provides access to the historical data, it is also necessary to track current user behavior and correlate it with the historical behavior, so that an appropriate response can be generated for online user interaction. We refer to the behavior representation of a user as a *dynamic profile*, since it captures changes in user behavior. Fast generation of dynamic profiles is difficult when a site is experiencing heavy user loads. To address this issue, we propose an online user interaction system that consists of a DataIndex-based data warehouse and a caching module. The data warehouse stores the information needed to create dynamic profiles and provides very fast access to this information. The caching module further improves the system performance by storing frequently requested profile information.

An implementation of an online user interaction system based on the proposed

framework is presented, along with a set of performance results which indicate that the system is indeed capable of providing near real-time responses, even under heavy user loads. Our performance results demonstrate the importance of an efficient data warehouse in the overall system to enable scalable online user interaction.

Chapter 1

Introduction

The growth of electronic commerce during recent years has resulted in new business strategies, as well as the emergence of new technologies to support these strategies. One such business strategy that has gained momentum recently is *mass customization* [105, 106]. The idea behind mass customization is to provide each individual consumer with products and services that are tailored to his or her preferences. While mass customization is very difficult to realize in the physical world, it is much more feasible with electronic commerce, since Web technologies allow content to be customized for users based on their preferences. In fact, mass customization originally referred to the physical modification of products and services to make them fit each consumer's needs [105]. More recently, mass customization has evolved to encompass a wide range of methods for customizing the *consumer experience* [106]. The consumer experience includes the physical products, which can be customized in function or in appearance, as well as the presentation of those products, which can be customized automatically or with the help of the consumer.

Web technologies play a key role in the consumer experience, primarily in terms of the presentation aspect. Web technologies allow the presentation of products to be "personally" designed for each consumer, based on certain criteria (e.g., user preferences, past purchasing behavior). A common theme of these technologies is that they attempt to provide some form of online interaction with users. Thus, throughout this dissertation, we refer to these technologies collectively as *online user interaction* technologies.

There are a number of reasons why online user interaction schemes have emerged. For instance, such technologies help consumers find the products they are looking for, reducing the information overload that consumers often experience. This in turn can convert browsers into buyers. Online user interaction schemes are also used to improve cross-sell by suggesting additional products for the customer to purchase. Another very compelling reason to use such technologies is to build customer loyalty. In e-commerce, where a site's competitors are only a click or two away, gaining consumer loyalty is an essential business strategy [113, 112].

Recent evidence suggests that online user interaction solutions can indeed have a significant business impact. For instance, Amazon.com is well known for its use of online customer interaction technology. In particular, Amazon's use of *recommendation technology* is believed to be the reason for Amazon's loyal customer base: 78% of Amazon's sales are from returning customers [45]. The business impact of online user interaction solutions is itself an interesting research problem, but is beyond the scope of this dissertation.

The basic idea behind virtually all online user interaction schemes is to accumulate vast amounts of historical data (usually stored in a database system) and then query this historical information based on "current" visitation patterns to provide a personalized experience. For example, suppose that analysis of an online bookseller's historical data were to suggest a strong link between interests in Science Fiction and Romance. Based on this knowledge, the bookseller might recommend a recent best-seller from the Science Fiction category to all shoppers browsing Romance books. Most current personalization software, such as NetPerceptions [104] or LikeMinds [9], offer the above-mentioned functionality, popularly known as recommendations. It so turns out that the technology underlying virtually all online recommendation engines is a clustering algorithm called *collaborative filtering* [124]. Essentially, collaborative filtering places users into static groups based on their preferences.

While such static profiling has indeed provided benefits, the static nature of this approach is problematic. For instance, consider the case where an e-shopper purchases a Chemistry textbook for his brother from an online bookseller, and later returns to the booksite looking for books for himself. After a few clicks, the system should recognize that the user is not interested in Chemistry books in his current visit, suppress its knowledge of his past behavior and adjust its responses to be in tune with his more recent behavior. This is possible only if the system can recognize changing behavior patterns. Thus, *true* online user interaction requires that profiles be dynamic, and therefore reduces to performing the following tasks:

1. Tracking users' movements or behavior patterns on a site
2. Accessing a vast knowledge base that correlates specific behavior with stored knowledge, and
3. Generating responses

A critical requirement is that the above tasks must be done in subsecond time frames.

The problem of providing such true online user interaction is, of course, one of scale. It is extremely difficult to track tens of thousands of online shoppers in near real-time, and even more difficult to access a database online and provide near real-time responses. This problem is only exacerbated as the number of online shoppers increases and as e-commerce sites attempt to collect more fine-grained customer information. The primary bottleneck lies in the underlying database systems - existing database systems cannot effectively support the requirements of true online user interaction. For this reason, existing online user interaction solutions rely on one of two basic approaches: (1) static profiling techniques that fit customers into one of a small number of predefined static profiles (usually based on statically pre-declared information, e.g., login or zip code) and provide canned online responses, or (2) delayed, offline interaction such as email or direct mail. The former approach is the approach taken by personalization software vendors, as described previously, while the latter approach constitutes a class of commercial software known as *customer relationship management* (CRM) systems. Clearly, both classes of solutions are not capable of meeting the online user interaction requirements outlined above.

1.1 Problems Addressed

This dissertation addresses the problems associated with providing scalable online user interaction. In particular, we address the following issues:

- *Scalable Data Warehouse Design.* A critical component of any online user interaction system is an efficient data warehouse. The warehouse must store large volumes of data and provide fast response times for queries on this data. The warehouse is typically vast, and may contain several types of data (e.g., navigational, transactional, demographic). Given that an e-commerce catalog may have virtually an infinite number of possible navigation paths, navigational data alone can easily result in warehouse sizes ranging from several hundred gigabytes to a few terabytes. The query operations required on this data will typically be complex, usually requiring joins of several large tables. In addition, given the interactive nature of the Web, query results must be delivered within subsecond time frames. Furthermore, the warehouse must be able to provide such performance even in the presence of heavy user loads (e.g., thousands to tens of thousands of simultaneous users).

- *Efficient Generation of Online Dynamic Profiles.* While the data warehouse provides access to the historical data in an online user interaction solution, it is also necessary to track current user behavior and correlate it with the historical behavior, so that an appropriate response can be generated by the application. We refer to the behavior representation of a user as a *dynamic profile*, since it captures changes in user behavior. Generation of dynamic profiles is difficult since these profiles must be generated quickly even when a site is experiencing heavy user loads.

We next elaborate on these problems.

1.1.1 Scalable Data Warehouse Design

A key component in any online user interaction solution is a data warehouse. The warehouse stores the historical customer data (e.g., navigational, transactional, demographic) that is used to generate responses. Since responses must be generated in subsecond time frames, the warehouse must be able to provide interactive query response times. There are several factors that make this a difficult task:

- The size of the warehouse is typically vast. Sites are collecting more and more data about their visitors. In particular, navigational data can easily add up to several hundred gigabytes or a few terabytes of data, especially when one considers the number of possible paths that a user can traverse through a Web site.

- The site will often experience high user loads. It is not uncommon for popular sites to have thousands to tens of thousands of simultaneous users at the site.

- The queries are typically ad-hoc and complex, involving joins of very large tables.

Thus, our objective in designing the data warehouse is to design a system which can provide fast query response times for complex, ad-hoc queries, even under high user loads.

1.1.2 Efficient Generation of Online Dynamic Profiles

A scalable data warehouse alone does not make a complete online user interaction solution. It is also necessary to track current user behavior and correlate it with the historical behavior, so that an appropriate response can be generated by the requesting application. We refer to the behavior representation of a user as a *dynamic profile*, since it captures changes in user behavior. A dynamic profile must be maintained for each and every visitor at a site. This requires that a dynamic profile be updated each time a user performs some action at the site (i.e., clicks). Furthermore, dynamic profiles must be generated very quickly (e.g., in sub-second time frames) so that the application can utilize the profile information to make a content delivery decision. For instance, a dynamic profile may indicate that a visitor to an online book store is about to leave the site without making a purchase. The book site can utilize this knowledge, along with other information contained in the visitor's dynamic profile, to make a special offer to the visitor to entice him to continue shopping at the site.

1.2 Contributions

This dissertation presents an approach for performing scalable online user interaction. The proposed approach allows highly scalable, online user interaction based on the real-time integration of a vast amount of extremely fine-grained historical data and online visitation patterns of e-commerce site visitors. The proposed framework comprises the following key components:

1. Observing specific instances of online behavior by tracking a large number of users (potentially tens of thousands) in real-time as they navigate through a site

2. Correlating this specific behavior with the vast amounts of historical behavior collected over time by performing retrievals from a large data warehouse in near real-time, and

3. Reacting to the user by delivering an appropriate user response, in near real-time, based on the system's knowledge of the user's current behavior and the information retrieved from the warehouse.

The proposed framework makes a research contribution in two broad areas:

1. Data Warehouse Design

2. Online User Interaction Modeling

We now expand on our contributions.

1.2.1 Data Warehouse Design

Our work contributes to the physical design of data warehouses, an area that is concerned with the data structures and algorithms used to store and retrieve data. Our contribution in this area is a new data storage and retrieval paradigm for data warehouses, referred to as *DataIndexes*. In addition to the DataIndex structures, two efficient algorithms for performing join queries with DataIndexes are presented. A mathematical analysis is presented which categorizes the cost of query evaluation for common classes of queries using DataIndexes. In addition, the results of an implementation based on DataIndexes is presented. Both the analysis and the implementation results indicate that DataIndexes significantly outperform existing indexing structures in many cases, and that they can provide the subsecond response times required for online user interaction.

1.2.2 Online User Interaction Modeling

Our contribution in this area is the design and implementation of a scalable online user interaction system which derives its scalability from an efficient data warehouse. We propose an online user interaction system that consists of a DataIndex-based data warehouse and a caching module. The data warehouse stores the information needed to create dynamic profiles and provides very fast access to this information. The caching module further improves the system performance by storing frequently requested profile information. An implementation of this system is presented, along with a set of performance results which indicate that the system is indeed capable of providing near real-time responses, even under heavy user loads. Our performance results also demonstrate the importance of an efficient data warehouse in the overall system.

1.3 Organization of the Dissertation

The remainder of this dissertation is organized into two parts: (I) *Scalable Data Warehouse Design*, and (II) *The Role of Data Warehousing in Enabling Scalable Online User Interaction*. Part I begins with an overview of data warehousing and a survey of the data warehousing literature that is relevant to this dissertation (Chapter 2). In Chapter 3, we present a novel physical design paradigm for data warehouses, including the *DataIndex* structures and a set of star join algorithms based on these structures. In Chapter 4, we present a set of cost analyses which compare the performance of the algorithms proposed in Chaper 3 to existing star join algorithms. We conclude

Part I with a discussion of an implementation of DataIndexes, along with a set of performance results in Chapter 5.

We begin Part II in Chapter 6 with an overview of online user interaction techniques and a survey of the relevant work in this area. This chapter also includes a discussion of several background concepts in online user interaction that are fundamental to the work in this dissertation. In Chapter 7, we discuss the design of the rule warehouse in an online user interaction system, including a comparative analysis of alternative designs. In particular, we compare DataIndex-based designs with conventional designs. In Chapter 8, we discuss the system architecture of an online user interaction system based on our proposed framework and the role of the rule warehouse in this system. In Chapter 9, we present the results of a set of experiments which demonstrate the performance of an online user interaction system which we have implemented. Our performance results focus on the impact of the rule warehouse in the overall system. Finally, we make concluding remarks in Chapter 10.

Part I

Scalable Data Warehouse Design

Chapter 2

Overview and Survey of Data Warehousing

We begin this chapter by providing a general overview of data warehousing. We then review the work in data warehousing that is relevant to this dissertation.

2.1 Overview of Data Warehousing

A data warehouse can be defined as a repository of historical data used to support decision making [73]. Online Analytical Processing (OLAP) refers to the technology that allows the user to efficiently retrieve data from the data warehouse. The characteristics of OLAP applications are quite different from those of operational or *On-Line Transaction Processing* (OLTP) systems. OLTP systems are designed to perform repetitive, structured tasks where detailed records are updated (e.g., order entry, account updates following a bank transaction). The emphasis in these systems is on maximizing transaction throughput and maintaining consistency. Typically OLTP systems are on the order of hundreds of megabytes to gigabytes in size.

In contrast to transactional database systems, data warehouses are designed for decision support purposes and contain long periods of historical data. For this reason, data warehouses tend to be much larger than transactional databases, often by orders of magnitude. It is quite possible for a data warehouse to be hundreds of gigabytes to terabytes in size [28]. In this environment, aggregated and summarized data are much more important than detailed records. The emphasis in data warehousing is on query processing and response times rather than transaction processing. Queries tend to be complex and ad-hoc, often requiring computationally expensive operations such as joins and aggregation. Further complicating this situation is the fact that such queries must be performed on tables having potentially millions of records. Moreover, the results have to be delivered interactively to the business analyst using the system.

From this brief overview, it is clear that data warehousing environments differ substantially from transactional or OLTP database systems. We now provide a more detailed discussion of data warehousing. We begin by discussing the architecture of a data warehousing system. This is followed by a discussion of data models for data warehousing/OLAP.

2.1.1 Data Warehousing Architecture

Figure 1 shows a typical data warehousing architecture.

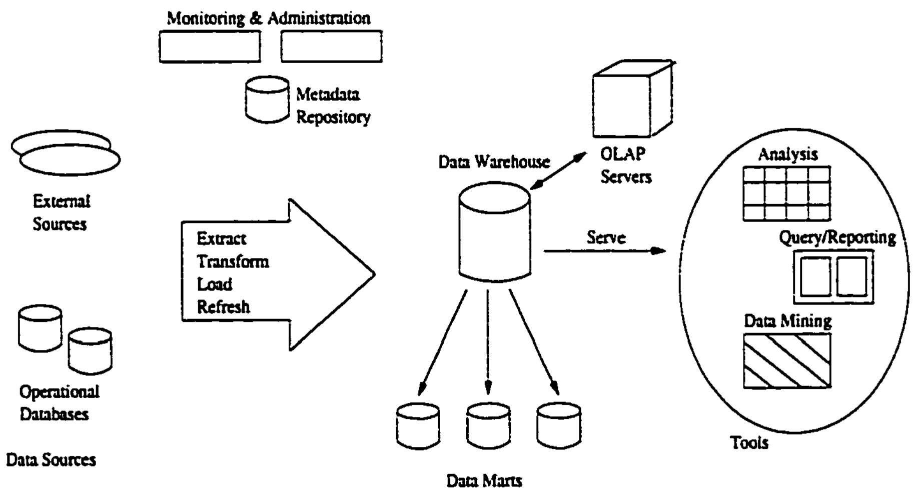


Figure 1: Data Warehousing Architecture

As the figure shows, the architecture includes tools for extracting data from multiple operational databases and external sources, for cleaning, transforming and integrating this data, for loading data into the warehouse, and for periodically refreshing the warehouse to reflect updates at the sources and to purge data from the warehouse (perhaps onto slower archival storage). In addition to the data warehouse, there may be several departmental data marts. Data in the warehouse and data marts is stored and managed by one or more warehouse servers, which present multidimensional views of data to a variety of front end tools, such as query tools, report writers, analysis tools, and data mining tools. Finally, there is a repository for storing and managing metadata, and tools for monitoring and administering the warehousing system.

We now discuss in more detail how the components of this architecture work. The first step is to extract data from the data sources. The data sources can be operational databases or other persistent storage (e.g., file systems), and can be local to the enterprise or foreign. Data extraction from such foreign sources is typically implemented via gateways and standard interfaces (e.g., ODBC [89]).

The next step is to "clean" the data. This step is necessary because of the fact that large volumes of data are combined from multiple data sources, thereby increasing the probability of errors and anomalies in the data. Examples of such errors and anomalies include inconsistent field lengths, inconsistent descriptions, inconsistent value assignments, missing entries and violation of integrity constraints. There are three broad classes of data cleaning tools:

- *Data Migration Tools.* This class of tools allows simple transformation rules to be specified. For instance, an example of such a transformation rule is "replace the string *work phone number* with *daytime phone number*".

- *Data Scrubbing Tools.* This class of tools uses domain specific knowledge to clean the data. They often employ parsing and fuzzy matching techniques to accomplish cleaning from multiple sources.

- *Data Auditing Tools.* This class of tools make it possible to discover rules and relationships by scanning data, similar to data mining tools. For example, such a tool may discover a suspicious pattern that a particular store location has never processed any refunds on an item.

Once the data has been extracted, cleaned, and transformed, the next step is to load it into the warehouse. Additional pre-processing may be done as part of this step, e.g., checking integrity constraints, sorting, aggregation, building indexes. Typically, batch load utilities are used for this purpose. Given the extremely large data volumes that must be loaded and the relatively small time window (usually at night) for loading, parallel loading techniques have been developed to speed up the loading process [13]. However, even using parallelism, a full load may still take too long. For this reason, most commercial utilities use *incremental loading* to reduce the volume of data that has to be incorporated into the warehouse. With incremental loading, only the updated tuples are inserted.

Periodically, the warehouse must be *refreshed* to reflect updates to the source data. There are two main sets of issues to consider: *when* to refresh and *how* to refresh. Usually, the warehouse is refreshed periodically (e.g., daily or weekly). The refresh policy is set by the warehouse administrator depending on the user needs and traffic, and may be different for different sources. Refresh techniques may also depend on the characteristics of the source and the capabilities of the database servers. Extracting an entire source file or database is usually too expensive, but may be the only choice for legacy data sources. Most modern database systems provide *replication servers* that support incremental techniques for propagating updates from a primary database to one or more replicas. Such replication servers can be used to incrementally refresh a warehouse when the sources change. In addition to propagating changes to the base data in the warehouse, the derived data (i.e., materialized views) must also be updated. The problem of constructing logically correct updates for incrementally updating materialized views has been the subject of much research (e.g., [31, 32, 151]). We will discuss this area of research in more detail later in this section.

An important ongoing activity is metadata and warehouse management. Warehouses typically contain multiple types of metadata. For instance, *administrative* metadata includes all the information necessary for setting up and using a warehouse (e.g., descriptions of source databases, warehouse schema definitions). *Business* metadata includes business terms and definitions, ownership of the data, and charging policies. *Operational* metadata includes information that is collected during the operation of the warehouse, such as the lineage of migrated and tranformed data and usage statistics. In addition to metadata management, there is also ongoing general warehouse management. Warehouse management tools are used for monitoring a warehouse and reporting statistics (e.g., query execution times, exception reporting).

Having discussed the architecture of a data warehousing environment, we now turn our attention to data models for data warehousing/OLAP.

2.1.2 Data Models for OLAP

The differing requirements of OLTP and OLAP systems dictate different data models for each type of system. In this section, we discuss conceptual, logical, and physical data models for OLAP.

2.1.2.1 Conceptual Models

The entity-relationship (ER) model is commonly used to represent an OLTP application at the conceptual level. However, this model is not well suited to the representation and efficient analysis of multidimensional data [77]. For this reason, an alternative conceptual model is required for OLAP systems. The *multidimensional data model* or *data cube* is a popular model used to conceptualize the data in a data warehouse [28].

The data cube contains points or "cells" that are *measures* or values based on a set of *dimensions*. For example, consider a retail sales application where the dimensions of interest may include CUSTOMER, PRODUCT, LOCATION, and TIME. If the measure of interest in this application is sales amount, then a point represents the sales measure corresponding to the CUSTOMER, PRODUCT, LOCATION, and TIME dimensions. In Figure 2, a data cube is provided which shows the PRODUCT, LOCATION, and TIME dimensions. A cell corresponds to the sales value for the corresponding PRODUCT, LOCATION, and TIME. For example, the shaded cell corresponds to sales for PRODUCT 'P1' in 'Seattle' for 1994. This representation is similar to the data model used for statistical databases, where the dimensions are actually *categories* and the measures are *summaries* [125].

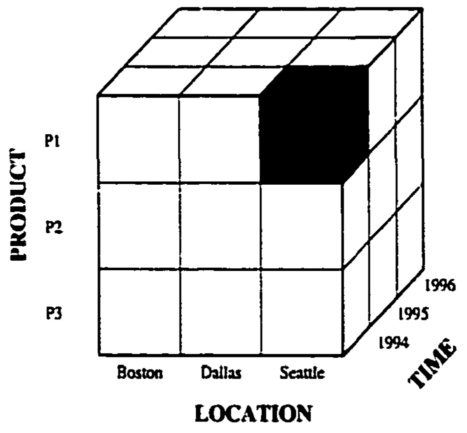


Figure 2: Data Cube for Sales Application

Dimensions often form a hierarchy. For instance, the TIME dimension may form a **day-month-year** hierarchy and the LOCATION dimension may form a **city-state-region** hierarchy. Dimensions allow different levels of granularity in the warehouse.

For example, `region` corresponds to a high level of granularity whereas `city` corresponds to a low level of granularity.

A number of data cube operations have been coined in the literature based on the multidimensional data model. These operations are described below:

Slicing refers to selecting the dimensions used to view the cube. Referring back to Figure 2, the dimensional view provided is PRODUCT by LOCATION with TIME in the background. With this view, for instance, an analyst may see all products for all locations for `1994`. This view can easily be changed using the slice operation. For example, a slice operation can change the view to PRODUCT by TIME with LOCATION in the background. With the resulting view, an analyst may then see all products for all years for `Boston`.

Dicing refers to selecting actual positions or values on a dimension. Selecting "Dallas" as the LOCATION is an example of dicing. Note that slice and dice together are (roughly) analogous to the relational algebra operators selection and projection, and have the effect of reducing the dimensionality of the cube.

Roll-up refers to increasing the level of granularity along one or more dimensional hierarchies. For example, consider again the sales cube of Figure 2, where the LOCATION dimension forms a `city-state-region` hierarchy. A typical analyst in an OLAP environment may wish to see the total sales at the `state` level, or at an even higher level, such as the `region` level.

Drill-down refers to decreasing the level of granularity and is the converse of roll-up. Drill-down is essential because an analyst often examines data first at an aggregated level and then selectively examines data in more detail. For example, suppose an analyst has total sales at the `region` level, but would like to see the corresponding totals for each `state`. In other words, the analyst would like to drill-down or move down the dimensional hierarchy. Again the analyst may desire even more detail and drill-down to the `city` level.

Pivot refers to aggregating using two or more grouping dimensions and producing a new multidimensional view of the data having an attribute for each grouping dimension and an additional attribute for the aggregated measure [28]. This operation is commonly found in multidimensional spreadsheet applications. Consider a simple example using the sales cube in Figure 2 where the dimensions are PRODUCT, LOCATION, and TIME and the desired aggregate measure is total sales by LOCATION and TIME. Hence, LOCATION and TIME are the grouping dimensions. Assuming the data values displayed in Table 1, the result of such a pivot would be a new view of the cube having LOCATION and TIME as dimensions, and total sales as a measure. A pivoted view is often displayed in a cross-tab format, as shown in Table 2. A result of the pivot operation is that values in the original cube become column headers (e.g., 1994) in the pivoted view.

Product	**Location**	**Time**	**Sales Amount**
P1	Boston	1994	100
P2	Boston	1994	200
P2	Boston	1995	200
P1	Dallas	1995	150
P2	Dallas	1995	150

Table 1: Sales Data for Pivot Example

Another distinctive feature of the conceptual model for OLAP is its stress on *aggregation* of measures by one or more dimensions as one of the key operations (e.g., computing and ranking the total sales by each county). Another popular operation is

Location	**1994**	**1995**
Boston	300	200
Dallas	0	300

Table 2: Result of Total Sales by LOCATION and TIME Pivot

comparing two measures aggregated by the same dimensions. In addition, time is a dimension that is of particular significance to data warehouses (e.g., trend analysis).

2.1.2.2 Logical Models

Logical data models have been proposed to support the data manipulation operations required in data warehousing/OLAP environments. Although there have been several proposals, no standard logical data model exists. An influential paper in this field appeared in 1995, written by Gray et al. [55]. In this paper, the authors propose the `CUBE` operator, which expands a relational table by computing the aggregations over all the possible subspaces created from the combinations of the attributes of such a relation. Since the appearance of this paper, a number of data models for OLAP have been proposed. All of the models proposed so far attempt to represent multi-dimensional databases, and are designed around the basic underlying construct of a **data cube**, containing *dimensions*, *attributes* and *measures*. These models be classified into two broad categories: (a) relational extensions, and (b) cube-oriented models.

- **Relational Extensions.** Relational extensions are based on the well known relational model [30]. In general, these models use relations to represent multi-dimensional constructs, such as dimensions. Work in this area includes [84, 62, 51].

- **Cube-Oriented Models.** Cube-oriented models attempt to model multi-dimensional databases more naturally. This does not mean that they are far from the relational paradigm - in fact all of them have mappings to it - but rather that their main entities are *cubes* and *dimensions*. Work in this area includes [4, 21, 143, 82, 137].

2.1.2.3 Physical Models

The underlying physical model supporting OLAP is centered around two major views. Whereas some vendors, especially vendors of traditional relational database management systems (RDBMS), advocate the *Relational OLAP* (ROLAP) architecture [90, 70], others support the *Multidimensional OLAP* (MOLAP) architecture [69]. The advantage of the MOLAP architecture is that it provides a direct multidimensional manipulation capability of the physical level of the data whereas the ROLAP architecture is just a multidimensional interface to relational data. On the other hand, the ROLAP architecture has two advantages: (a) it can be easily integrated into other existing relational information systems, and (b) relational data can often be stored more efficiently than multidimensional data.

In the ROLAP approach, the data is stored in a relational database using a special schema instead of a traditional relational design. The highly normalized form of data advocated by conventional design methodologies is inappropriate in an OLAP environment for performance reasons. A high degree of normalization entails a large number of joins, which greatly increases response times, especially given the size of most data warehouses. For this reason, a special schema known as the *star schema* (or its variants the *snowflake* and the *constellation schema*) is often used. A star schema usually consists of a single *fact table* and a *dimension table* for each dimension. The fact table contains foreign keys to each dimension table, along with the actual metric data (e.g., sales amount). An extended version of the star schema, the *snowflake schema*, is often used to represent the dimensional hierarchies in normalized form.

A possible snowflake schema for the sales application is presented in Figure 3. This figure displays the `day-month-year` and `city-state-region` hierarchies as somewhat normalized.

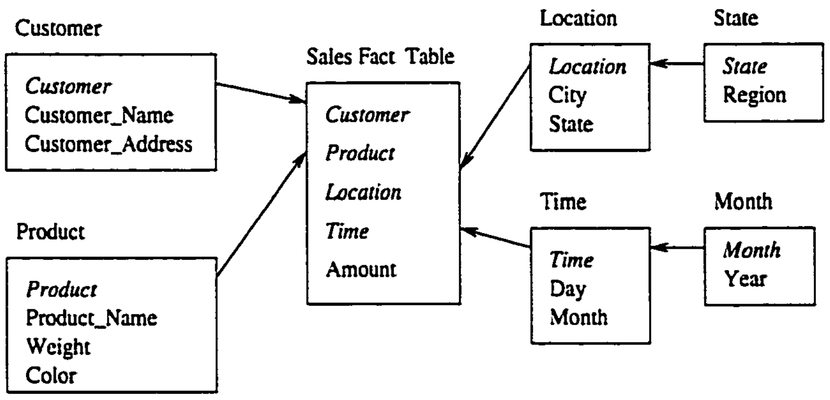


Figure 3: Snowflake Schema for Sales Application

In the MOLAP architecture, data is stored in n-dimensional arrays. Systems following this approach are referred to as *multidimensional database systems* (MDBMS). Each dimension of the array represents the respective dimension of the cube. The contents of the array are the measure(s) of the cube. An MDBMS requires the precomputation of all possible aggregations. Thus, they often perform better than traditional RDMBS [33]. However, they are typically more difficult to update and administer.

Having provided an overview of data warehousing/OLAP, we now turn our attention to the work in this area that is most relevant to this dissertation. In particular, we review the work that has been done to improve query performance in data warehouses.

2.2 Efficient Query Processing in Data Warehouses

Given the characteristics of data warehousing/OLAP environments, it is clear that the emphasis in OLAP environments is on efficient query processing. A number of "conventional" relational query processing approaches have been applied to or extended for answering OLAP queries. Some of this work has concentrated on efficiently performing GROUP BY [24, 26, 52], aggregation [27, 57, 68, 65, 109, 148, 150], join or range queries [67, 130, 141], or supporting incomplete query answers [22, 64, 145]. Several approaches have been proposed for supporting the SQL CUBE operator, including [3, 44, 57, 93, 116, 126].

The majority of the work in attempting to improve query performance in data warehouses can be classified into two broad categories: (a) *precomputation strategies*, and (b) *ad-hoc strategies*. We describe both areas of work in the sections that follow.

2.2.1 Precomputation Strategies

Precomputation strategies rely on *summary tables* or *materialized views*, i.e., derived tables that house precomputed or "ready-made" answers to queries [28]. This has been, by far, the most explored area in the context of data warehouses [2, 31, 56, 59, 60, 58, 63, 85, 92, 107, 108, 151]. The basic premise underlying this work is that data warehouses can achieve faster response times by pre-aggregating (i.e., materializing) the answers to frequently asked queries. The main challenges facing researchers in this area include: (a) identifying the views to materialize, (b) exploiting the materialized views to answer queries, and (c) efficiently updating the materialized views during load and refresh.

The selection of views to materialize must take into account workload characteristics, the costs for incremental update, and upper bounds on storage requirements.

Under simplifying assumptions, a greedy algorithm was shown to have good performance [63]. A recent work proposed a system referred to as *DynaMat* [79], which dynamically materializes information at multiple levels of granularity in order to match the workload, but also takes into account the maintenance restrictions for the warehouse, such as downtime to update the views and space availability. The problem of exploiting materialized views to answer queries has been addressed by query optimization work, such as [25, 83]. Finally, efficiently updating materialized views is an area that has seen significant work (e.g., [93, 108, 80, 149]).

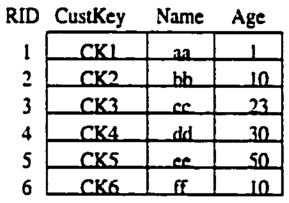
It is recognized however, that the anticipatory approach used in precomputation strategies only works up to a point [28, 98], and a considerable fraction of the workload in OLAP applications consists of *ad-hoc* queries which need to be computed on demand [10]. This has led to work on strategies for ad-hoc query processing, described next.

2.2.2 Ad-hoc Strategies

Ad-hoc strategies support ad-hoc querying of OLAP data by using fast access structures on the base data. Database systems use indexes to improve efficiency of access to data. Various general purpose indexing techniques have been proposed and are utilized in OLTP systems, including hashing [118], B trees [29, 34], and multidimensional trees such as the R-tree [61], the K-D-B tree [115], the BV tree [49], and the X-tree [14]. There exists another class of multidimensional structures, namely *grid files* [95], that allows for very fast access to multidimensional data.

For data warehousing/OLAP systems, an influential paper by O'Neil and Quass [98] identified four index structures that have been shown to improve query processing performance for certain classes of queries in data warehousing environments: *B*$^+$-*trees* [34], *bitmapped indexes* [96, 99], *bit-sliced indexes* [98, 72], and *projection indexes* [98, 133]. These structures are further described below.

- *B*$^+$-*trees* . B$^+$-trees [34] have been used extensively in OLTP systems. The B$^+$-tree is a multi-level index structure, where the leaf level contains a list of pointers or *row identifiers* (RIDs) for each value in the indexed column. For example, consider the **CUSTOMER** table shown in Figure 4. The RID column shown in the figure represents pointers to the locations of the tuples on disk.



RID	CustKey	Name	Age
1	CK1	aa	1
2	CK2	bb	10
3	CK3	cc	23
4	CK4	dd	30
5	CK5	ee	50
6	CK6	ff	10

Figure 4: Example Customer Table

A B$^+$-tree index on the **Age** column is shown in Figure 5[a]. The leaf level of the index contains a list of pointers or RID list for each unique value in the **Age** column. B$^+$-trees are used in most RDBMSs.

- *Bitmapped Indexes.* Bitmapped indexes [96] are similar to B$^+$-trees , except that the leaf level contains pointers to *bit vectors*, rather than RID lists. A bit vector has the cardinality of the indexed column, and a bit is set if the value in the table row matches the value in the index. Figure 5[b] shows a bitmapped index for the **Age** column. Bitmapped indexes are used in the Oracle family of database products (e.g., Oracle 8i,9i) [99, 101].

- *Projection Indexes.* A projection index [98] is simply a copy of a column that is used as an index. Note that some implementations of projection indexes do

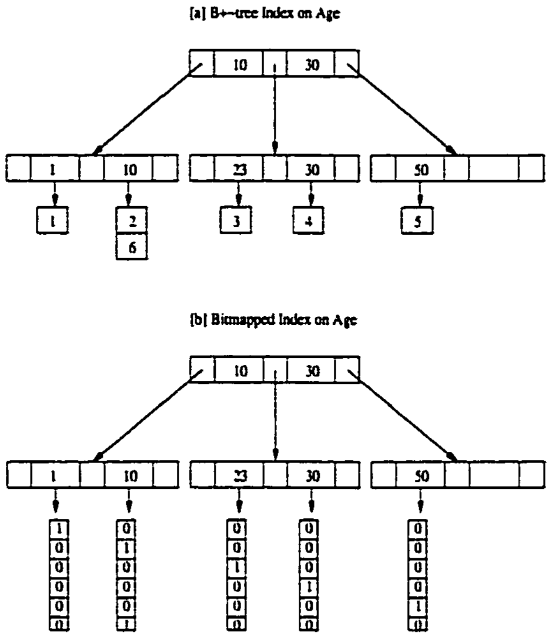


Figure 5: B$^+$-tree and Bitmapped Indexes

not store a separate copy of the indexed column. Projection indexes are used in the Sybase IQ [133] product.

- *Bit-sliced Indexes.* A bit-sliced index [98] is a "bit-level" projection index. In other words, a bit-sliced index is a *bitwise* vertical partition of a projection index. Bit-sliced indexes are suitable for numeric (fixed-point) or ordinal attributes. Bit-sliced indexes are used in the Informix [72] product.

There has been some additional work related to bitmapped indexing techniques. An efficient encoding scheme for bitmapped indexes is presented in [146], which attempts to reduce the size of bitmapped indexes. While this technique appears to be promising, there are still many open issues, such as how to determine the proper encoding. In [23], a framework for the design and evaluation of bitmap indexing schemes is proposed.

More recently, there have been a few new index structures proposed specifically for data warehousing/OLAP. The *Cubetree* [116], a collection of packed R-trees [61, 117], is a multidimensional indexing scheme for the OLAP data cube. The Y-tree [74] is a tree-based index for data warehouses based on the notion of bulk insertion, and thus, the emphasis is on allowing fast insertions. The DC-tree [47] is a multidimensional index similar to the X-tree [14], and the emphasis of this work is also on efficiently updating the index.

The index structures described so far map column values to their containing rows in a single table. There exists a class of index structures known as *join indexes* [139], which typically associate column values and rows of two tables. In this way, the join index represents the fully precomputed join. Based on this idea, [97] introduced the *bitmapped join index* to improve the performance of join queries in data warehouses. The bitmapped join index is similar to the bitmapped index, except that it stores pointers to the referenced table rows rather than data values. Bitmapped join indexes are used in certain commercially available products, such as Informix [72]. We will discuss this structure in greater detail in a later chapter.

Having discussed the work that is relevant to this dissertation, the next several chapters will focus on the design of a scalable data warehousing solution approach. In particular, the next chapter will present a novel physical design paradigm for data warehouses, based on a suite of structures referred to as *DataIndexes*. DataIndexes combine and extend, in an effective way, ideas embedded in other well-known database structuring techniques, specifically *vertical partitioning* and *transposed files* [125], as well as indexing techniques, specifically projection indexes [98] and bit-mapped join indexes [97].

Chapter 3

Physical Data Index Design: The DataIndex

This chapter presents a novel physical design paradigm for data warehouses which is based on a suite of structures referred to as *DataIndexes*. The DataIndex structures are first presented, followed by a set of star-join algorithms which are based on these structures.

3.1 Motivation behind DataIndexes

In Chapter 2, we described several index structures, such as B trees and bitmapped indexes . These index structures all have one common characteristic: they each prescribe a separate set of indices or access structures *in addition to the base data.* Given the large size of data warehouses, storage is a non-trivial cost, and so is the additional storage requirement due to the index structures. This is especially true given that data and storage maintenance costs are often up to seven times as high per year as the original purchase cost [128]. Hence, a terabyte-sized system, with an initial media cost of $100,000, could cost an additional $700,000 for every year it is operational. This cost is certainly non-trivial. Indexes, obviously, add to this cost and hence it is essential to minimize these additional costs. Unfortunately, as we will soon show, even the simplest index structure used today incurs substantial increases in total storage requirements, both in absolute and percentage terms. This, in turn, translates into higher media and maintenance costs. More importantly though, intuition dictates that an increased overall database size should result in lower performance. This prompts us to ask the following question: "*is it possible to reduce storage requirements, without sacrificing the efficiency obtained from indexing?*"

In this chapter, we answer the above question in the affirmative by proposing *DataIndexes* as a novel paradigm for storing the base data *as well as* serving as access structures to this data in warehouses. Because DataIndexes are both storage and access structures, substantial space savings are realized.

Thus, DataIndexes are motivated by the desire to reduce the additional storage costs due to index structures. To illustrate these costs we refer to the star schema [76] presented in Figure 6, which was derived from the TPC-D benchmark database [138] with a *scale factor* [1] of 1. This schema models the activities of a world-wide wholesale supplier over a period of 7 years, and will be used as a running example throughout this chapter. The central *fact* table is the SALES table, and the dimensions of the data are captured through the PART, SUPPLIER, CUSTOMER and TIME tables. Each dimension table has a primary key. The fact table is associated, through foreign-key reference, to the four dimension tables. Note that some applications do not enforce referential integrity between the fact and dimension tables. However, we assume throughout the chapter that *referential integrity is strictly enforced in all star schemas.*

We first compute the storage requirements for the data, based on a conventional relational implementation, where the five tables would be represented as a series of records partitioned over a set of data blocks. To simplify our analysis, we assume that each record is small enough to fit within a data block. Given the usually large sizes ($\geq$ 8 KB) of warehouse data blocks, this assumption is very realistic. We also assume

[1] In the TPC-D benchmark, the scale factor is used to define the size of the database. In TPC-D, a scale factor of 1 corresponds to a database size of approximately 1 GB, the minimum required size for a test database.

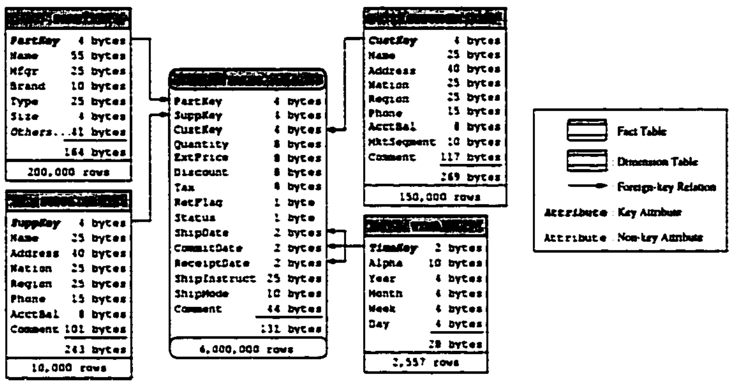


Figure 6: A Sample Warehouse Star Schema

that each block contains some header information (e.g., version number, pointers to other blocks, etc.) which makes its *effective size*, B, smaller than its actual size, B_{act}. Given a particular table, T, with a record width $w(T)$ and cardinality $|T|$, one can compute the number of records that fit in a data block, i.e., the *blocking factor* for T, as $\rho(T) = \left\lfloor \frac{B}{w(T)} \right\rfloor$. In turn, we determine the size of the table to be $B_{act} \times \left\lceil \frac{|T|}{\rho(T)} \right\rceil$. Let us further assume that the implementation platform uses a block size (B_{act}) of 8192 bytes and an effective block size (B) of 8000 bytes. From this we compute the size of each table as shown in Table 3.

Table	**Table Size** (bytes)
SALES	805,773,312
PART	34,136,064
SUPPLIER	2,564,096
CUSTOMER	42,377,216
TIME	73,728
all tables	884,924,416

Table 3: Table Sizes For Example Star Schema

We next compute the total storage requirements of the sample database based on the exact expressions for the size of the various index types derived in [144]. We have selected four popular index structures for data warehousing : *B-trees* [34], *bitmapped indexes* [96, 99], *bit-sliced indexes* [98, 72], and *projection indexes* [98, 133]. It is assumed that only the PartKey, SuppKey, CustKey, ShipDate, CommitDate and ReceiptDate columns of the SALES table are indexed. Table 4 summarizes the storage requirements of each indexing scheme for the sample database[2]. Table 4 indicates

Indexed Column	B^+-tree	Bitmapped	Projection	Bit-sliced
PartKey	1,675,288,576	1,640,816,640	24,576,000	24,649,728
SuppKey	118,800,384	82,780,160	24,576,000	24,649,728
CustKey	1,265,680,384	1,230,807,040	24,576,000	24,649,728
ShipDate or CommitDate or ReceiptDate	36,896,768	21,733,376	12,288,000	12,324,960
total	3,170,459,648	3,019,603,969	110.592,000	110,936,064

Table 4: Indexing Costs for Example Star Schema

that the the indexing overhead for projection indexes is the least; the other access structures yield higher storage overheads. But even projection indexes incur a more than 12.5% increase in size of the indexed database over the unindexed database, assuming the index structures are stored in addition to the data. (Some implementations of projection indexes, such as Sybase IQ [133], do not store both the index and data.) We also emphasize that projection indexes are not very effective for many OLAP queries. While they may perform well for simple queries involving a single table (e.g., restrictions, counting), they do not offer any improvement over conventional pairwise join techniques [54]. Typically, additional indexes are required along with projection indexes to improve join performance, incurring additional overhead.

Clearly, there is a need to minimize the additional overhead incurred by index structures. This is the motivation behind DataIndexes, introduced next.

[2]Note that the sizes of both the standard and bitmapped B-trees depend on the distribution of data values. The numbers presented in table 4 correspond to a perfectly uniform distribution of values.

3.2 The DataIndex

In this section, we propose the *DataIndex*, which is a storage structure that serves both as an index as well as data. Specifically, we examine two types of DataIndexes, both based on the same basic idea of vertical partitioning. We refer to these as *Basic DataIndexes* and *Join DataIndexes*. We then compare Dataindexes to existing indexing approaches and then discuss physical database design based on DataIndexes.

3.2.1 Basic DataIndex (BDI)

A DataIndex is simply a vertical partition of a relational table. In this sort of partitioning, the columns being indexed are removed from the original table, and stored separately, with each entry being in the same ordinal position as its corresponding base record. The isolated partitions can then be used for fast access to data in the table. We call this partition a *Basic DataIndex* (BDI). A graphical representation of this structure is shown in figure 7.

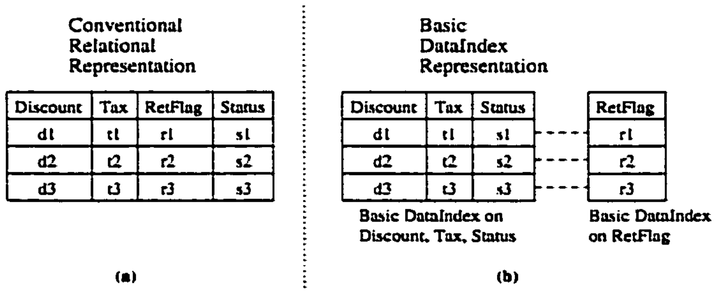

Conventional Relational Representation
Discount | Tax | RetFlag | Status
d1 | t1 | r1 | s1
d2 | t2 | r2 | s2
d3 | t3 | r3 | s3
(a)
Basic DataIndex Representation
Discount | Tax | Status
d1 | t1 | s1
d2 | t2 | s2
d3 | t3 | s3
RetFlag
r1
r2
r3
Basic DataIndex on Discount, Tax, Status
Basic DataIndex on RetFlag
(b)

Figure 7: The Basic DataIndex

In this figure, we show the actual storage configurations of the two cases: a base table (Fig. 7a) and the corresponding BDIs (Fig. 7b). The base table consists of the attributes Discount, Tax, RetFlag, Status, and two BDIs are constructed; one on the RetFlag column, and the other on the Discount, Tax and Status columns.

As indicated by the dotted lines joining records from the two BDIs, the order of the records in the base table is conserved in both DataIndexes. This allows for an easy mapping between entries in the two BDIs. This mapping relies on the fact that, for a given BDI, there exists a fixed number of "slots" for holding records in each data block. Denoting by $w(\beta)$ the width of a record of BDI β, the number of slots per page in the BDI is given by $\rho(\beta) = \left\lfloor \frac{B}{w(\beta)} \right\rfloor$.

Based on this value, one can simply associate the elements of a record in the two BDIs through a simple arithmetic mapping. Assume for instance that we need to access the record in the base table corresponding to some **RetFlag** value. This translates to associating the corresponding **RetFlag** BDI record r with its matching record in the other BDI. Denoting the BDI on **RetFlag** by β_R and the BDI on **Discount**, **Tax**, and **Status** by β_{DTS}, we first compute the RID of record r in β_R, i.e., we determine that r is located in the $(b_R)^{\text{th}}$ block of β_R, at slot number s_R. This can be done by simply keeping track of the number of data blocks loaded during the scan operation and seeing the position of the matching record within its data block. The ordinal position, Ω, of this record in the base table is simply given by $\Omega = b_R \times \rho(R) + s_R$. From this number, we can determine the RID of the corresponding entry in the other BDI (β_{DTS}). This RID is characterized by b_{DTS}, the ordinal number of the β_{DTS} data block in which the record is located, and s_{DTS}, the slot in block b_{DTS} corresponding to record r. This RID can be expressed as $b_{DTS} = \left\lfloor \frac{\Omega}{\rho(R)} \right\rfloor$, and the slot number as $s_R = \Omega \bmod \rho(R)$.

It is the *ordinal mapping* that makes this basic approach more efficient than existing vertical partitioning methods such as the Decomposition Storage Model (DSM) [36, 75, 140]. Indeed, DSM utilizes surrogate keys to map individual attributes together, hence requiring a surrogate key to be associated with each attribute of each record in the database.

The resulting database size is essentially the same as the size of the raw data in the original database configuration. However, we can now utilize the separate dimensional columns of the partitioned fact table as both elements of and indexes onto that table. Through this simple technique, we can store the data and index for the same storage cost as for the data alone. Hence, in terms of storage, the indexing is free.

Turning to our running example, we can divide the SALES table in Fig. 6, into five smaller tables, as shown in Fig. 8. The new schema is then composed of 5 vertical

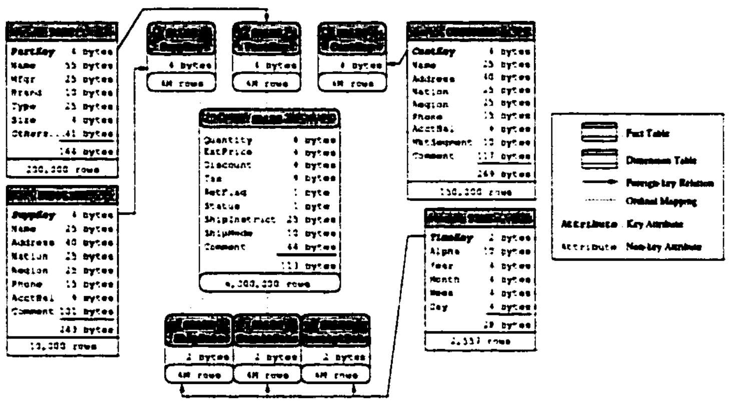


Figure 8: Example Warehouse Schema with DataIndex

partitions: one for each of the SuppKey, PartKey, and CustKey dimensional attributes, one for the combination of ShipDate, CommitDate, and ReceiptDate dimensional attributes and one for the remaining columns from the original SALES table. A record in the original SALES table is now partitioned into 5 individual records, one in each of the resulting tables. Any such record can easily be re-built from these, since its component rows in the 5 resulting tables all share the same *ordinal position.* In the example, each of the 5 new tables is a DataIndex.

We note that a potential problem arises in the presence of variable-length attributes (e.g., those of type VARCHAR). In such cases, the number of records can vary from one page to the next. To solve this problem, one can define a maximum number of records per page, as is done in the model 204 database system [96]. In this case, a few ordinal position numbers (Ω) may not actually correspond to actual records. Alternatively, one can "encode" each unique value to a fixed length surrogate. To simplify our analysis, in this paper we thus assume that field lengths are fixed with no loss of generality.

3.2.2 Join DataIndex (JDI)

In decision support databases, a large portion of the workload consists of queries that operate on multiple tables. Many queries on the star schema of Fig. 6 would access one or more dimension tables and the central SALES table. For instance, a marketing analyst might want to identify the part type most often purchased by different customer groups identified by their nation and market segment. The PART, CUSTOMER and SALES table must be joined to answer this query. Access methods that efficiently support join operations thus become crucial in decision support environments [97, 110]. The idea of a BDI presented in the previous section can very easily be extended to support such operations. Consider for instance, an analyst who is interested in possible trends or seasonalities in discounts offered to customers. This analysis would be based on the following query:

```
  SELECT  TIME.Year, TIME.Month, average(SALES.Discount)
    FROM  TIME, SALES
   WHERE  TIME.TimeKey = SALES.ShipDate
GROUP BY  TIME.Year, TIME.Month
```

Using the conventional relational approach, the association between the two tables TIME and SALES in Fig. 6 is implemented through the primary key/foreign key relationship linking the columns ShipDate and TimeKey. To perform a join operation on these two tables, the two columns must be accessed to determine the records that are join candidates. There exist relatively fast algorithms (e.g., merge and hash joins) for evaluating joins. However, approaches that use pointers to the underlying data, instead of the actual records, tend to give a better performance than other join strategies [54]. Thus, if a DataIndex relied on pointers to records to both store and index the underlying data, it would perhaps have a good join performance.

Indeed, one can significantly reduce the number of data blocks to be accessed while processing a join by storing the RIDs of the matching records in the corresponding dimension table – instead of the corresponding key values – in a BDI for a foreign key column. This structure is a *Join DataIndex* (JDI). The JDI on `SALES.ShipDate` would then consist of a list of RIDs on the `TIME` table. Such a JDI is shown in Fig. 9. As before, we show both the conventional relational and our proposed representations. In the conventional approach, we show referential integrity links between the `SALES` and `TIME` tables as dashed arrows. For our proposed approach, we use solid arrows to show the rows to which different RIDs point and dotted lines to show that the order of the records in the JDI and the `SALES` BDI is preserved from the base table.

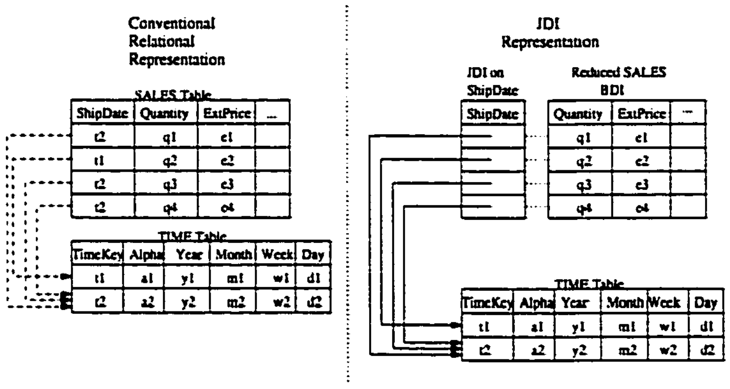


Figure 9: The Join DataIndex

As can be seen in this figure, instead of storing the data corresponding to the `ShipDate` column, the JDI provides a *direct* mapping between individual tuples of the `SALES` and `TIME` tables. The join required to answer the above query can thus be performed in a single scan of the JDI (more details in section 3.3). This property of JDIs is indeed attractive, since the size of this index is, of course, proportional to the number of tuples in the table from which it was derived. In our example schema, for instance, the JDI on ShipDate contains 6 million entries. A join operation could thus be performed by examining each one of these entries in turn. This approach should be significantly faster than with conventional join algorithms, which typically perform joins between two or more tables in a pairwise fashion. Such algorithms include nested-loop joins, merge joins [16], hash-joins [17], or any derivative of these techniques [78, 123] (see [54] for a survey). In fact, the exact number of block accesses needed to scan a JDI is simply the number of data blocks occupied by this structure. This is given by $\left\lceil \frac{|\mathrm{SALES}|}{\rho(r)} \right\rceil$, where r is the size of a RID (6 bytes). This results in $\left\lceil \frac{6{,}000{,}000}{\left\lfloor \frac{8000}{6} \right\rfloor} \right\rceil = 4502$ block accesses.

However, a JDI does not contain any data values. It might thus make the evaluation of queries where these values are needed more difficult. For instance, consider the following query:

```
SELECT ShipDate FROM SALES
```

If the ShipDate column is stored as a JDI, this query requires access to both the SALES and TIME tables, even though the latter one is not explicitly specified. We can thus compute the number of block accesses necessary to evaluate the above query as $4502 + \left\lceil \frac{|\mathrm{TIME}|}{\rho(\mathrm{TIME})} \right\rceil = 4511$. This figure is somewhat larger than would be required with a BDI. With a BDI, evaluating the above query would require only $\left\lceil \frac{|\mathrm{SALES}|}{\rho(\mathrm{ShipDate})} \right\rceil = 1500$ block accesses.

However, this would not always be the case: if the foreign table is small and the width of the indexed column is large, then scanning it to obtain data values will be more efficient than actually scanning a corresponding BDI. For instance, consider the (somewhat unlikely) case where a TimeKey value is 8-bytes wide (i.e., $w(\mathrm{TimeKey}) = 8$). Evaluating the above query with a JDI would require $4502 + \left\lceil \frac{|\mathrm{TIME}|}{\rho(\mathrm{TIME})} \right\rceil = 4513$ block accesses, where $\rho(\texttt{TIME})$ is now $\lfloor \frac{8000}{34} \rfloor$, while a BDI would require $\lceil \frac{|\text{SALES}|}{\rho(\text{TimeKey})} \rceil = 6000$ I/O operations, where $\rho(\texttt{TimeKey})$ is now $\lfloor \frac{8000}{8} \rfloor$. There would thus appear to be situations where a JDI is, in fact, preferable to a BDI even when no join is explicitly involved[3]. Hence, even though a JDI is useful for a column storing foreign keys, it is also useful when the column is wide and the number of distinct values in the column is small. In this case, it is preferable to realize the column as a JDI with the addition of another (small) lookup column storing the distinct values in the original columns. When it pays to do this is precisely characterized in section 4.2.1.5.

In Sections 4.2.1.1 and 4.2.1.5 we perform an analysis of the performance of the different indexing schemes in order to characterize exactly when a particular indexing scheme is to be preferred. Now we provide a qualitative summary of the features embedded in DataIndexes.

3.2.3 Comparison of BDIs and JDIs with existing Indexing Approaches

The Basic DataIndex is closely related to the idea of the Projection Index. A projection index is simply a mirror image of the column being indexed. When indexing columns of the fact table, storing both the index and the corresponding column in the fact table results in a duplication of data. In such situations, it is advisable to only store the index if original table records can be reconstructed easily from the index itself. This is the starting point of the proposed DataIndex scheme and is how

[3]Interestingly, we can significantly reduce the cost of accessing the TIME table in this sort of query by storing its primary key column (i.e., TimeKey) as a BDI. In this case, the number of I/Os required to evaluate the above query with a JDI is equivalent to the number of blocks of *both* the JDI on SALES.ShipDate and the BDI on TimeKey. As it turns out, the TIME table is so small that only 3 blocks are required to store the BDI on TimeKey. Hence, in this particular example, the total number of block accesses required to evaluate the above query is only 4505, which is again smaller than would be required with a regular BDI (whose size, in this case, was computed above to be 6000 blocks).

Sybase IQ stores data [50, 98]. Furthermore, with DataIndexes, each BDI of a table is stored separately – with ordinal positon based mapping providing more efficient access to individual record fields compared to other vertical partitioning based methods. Because BDIs are stored separately, only columns of interest need to be loaded in memory when joins are performed. Another attractive aspect of BDIs, and a point of departure from pure projection indexes, is that each BDI can contain any number of columns from the original table, unlike projection indexes which are restricted to single columns. Finally, as mentioned previously in Section 3.1, projection indexes do not improve join performance. We introduce the notion of *Join DataIndexes* (JDI) for this purpose.

O'Neil and Graefe [97] briefly introduced the idea of a bitmapped join index for efficiently supporting multi-table joins. JDIs capitalize on this idea in the context of the Basic DataIndex. A bitmapped join index (BJI) associates related rows from two tables [97], as follows. Consider two tables, T_1 and T_2, related by a one-to-many relationship (i.e., one record of T_1 is referenced by many records of T_2). A BJI from T_1 to T_2 can be seen as a bitmapped index that uses RIDs of T_1 instead of search-key values to index the records of T_2. Using a similar basic philosophy, a JDI stores the RIDs of the matching records in the corresponding dimension table instead of the corresponding key values. There are however, two important differences between JDIs and BJIs implemented in commercial systems:

1. Commercial implementations of BJI (such as in INFORMIX) are tree structured and exist as separate index structures. JDIs are flat structures and *do not exist* as auxiliary index structures. Rather, JDIs are a representation of the base data itself.

2. As we saw, JDIs are useful even when no joins are performed, specifically, when a column stores foreign keys, it is useful when the column is wide and the number

of distinct values in the column is small. This implies that with DataIndexes, *the amount of storage required may even be smaller than the storage required for the base tables.*

We discussed above how DataIndexes are different from the indexes proposed for warehouses so far. In this context we must also mention that the way *we use* these structures is also radically different from how current structures are employed. More specifically, we design a number of query processing algorithms that use DataIndexes in novel ways to deliver much improved performances of star-join queries in a large number of cases.

In summary, DataIndexes take the best aspects of vertical partitioning, projection indexes, and Bitmapped Join Indexes and integrates as well as extends them in (what we will show to be) an effective manner.

Before we conclude this section, it is important to point out that a number of *variant* indexes are supported in commercial products such as Sybase IQ [133], Oracle 8 [99], Informix Universal Server [72], and Red Brick Warehouse [110]. In addition to projection indexes[98] and bitmapped join indexes [97] mentioned already, such index structures include bitmapped indexes [96], bit-sliced indexes[98]. An analysis of three index structures along with B^+-trees is presented in [98], which indicates that these four structures are particularly appropriate for warehousing/OLAP environments. In Section 4.2, we present the results of extensive analysis of the previously proposed index structures along with DataIndexes.

3.2.4 The DataIndex Physical Design Strategy

Having introduced the two types of DataIndexes, we now briefly describe a physical design strategy based on these indexing structures. In general, we propose that a JDI be established for each foreign column in the fact table, and single-column BDIs be established for all other fact table columns and for all dimension table columns. Thus every column in a given star schema is represented as either a single-column BDI or as a JDI.

To illustrate, consider again the star schema of Figure 6. The foreign columns in the SALES fact table are PartKey, SuppKey, CustKey, ShipDate, CommitDate, and ReceiptDate. If we let j_A denote a JDI on attribute A in the SALES fact table, then our physical design strategy assumes the following JDIs are defined: $j_{\texttt{PartKey}}$ corresponding to the PART dimension table, $j_{\texttt{SuppKey}}$ corresponding to the SUPPLIER dimension table, $j_{\texttt{CustKey}}$ corresponding to the CUSTOMER dimension table, $j_{\texttt{ShipDate}}$, $j_{\texttt{CommitDate}}$, and $j_{\texttt{ReceiptDate}}$ all corresponding to the TIME dimension table. The remaining columns in the fact table would be stored as BDIs. For instance, single-column BDIs would be defined for SALES.Quantity, SALES.ExtPrice, and SALES.Discount. Likewise, single-column BDIs would be defined for all dimension table columns. These would include, for instance, TIME.TimeKey, TIME.Alpha, TIME.Year, TIME.Month, TIME.Week, and TIME.Day for the TIME dimension table. The physical design strategy we have described will be assumed throughout this paper.

Having described the DataIndex structures, we now proceed to describe a set of star-join algorithms which are based on the DataIndex structures.

3.3 Fast Star Join Algorithms Based on DataIndexes

A common operation in OLAP applications is the *star-join* query. In a star-join, the fact table is joined with a set of dimension tables. Due to the large size of most data warehouses, a star-join is typically an extremely expensive operation. As mentioned previously, response time is critical in OLAP applications. Therefore, it is imperative to have algorithms that can perform star-join queries very quickly. Such algorithms must ensure that the appropriate access structures are utilized. In this section, we present efficient algorithms for performing star-join operations with DataIndexes.

A typical OLAP query is of the form

SELECT column list FROM F, D_1, ..., D_m WHERE $P_{\bowtie}$ AND P_σ

where F is the central fact table, $D_1, \ldots, D_m$ are the m dimensional tables participating in the join, P_σ is a set of *selection* predicates (i.e., each individual predicate only concerns one table), and $P_{\bowtie}$ is a set of *join* predicates (i.e., each predicate is of the form $F.A_1 = D_i.A_2$). To illustrate, consider the following query, based on our example from section 3.2, which lists the prices for sales made locally by suppliers in the United States:

```
SELECT U.Name, S.ExtPrice
FROM SALES S, TIME T, CUSTOMER C, SUPPLIER U
WHERE T.Year BETWEEN 1996 AND 1998 AND U.Nation='United States' AND
C.Nation='United States'
AND S.ShipDate = T.TimeKey AND S.CustKey = C.CustKey AND S.SuppKey =
U.SuppKey
```

In this query, the selection predicates (i.e., P_σ) are "`T.Year BETWEEN 1996 AND 1998 AND U.Nation='United States' AND C.Nation='United States'`"; the joining predicates ($P_{\bowtie}$) on the other hand are "`S.ShipDate = T.TimeKey AND S.CustKey=C.CustKey AND S.SuppKey = U.SuppKey`". We will utilize this query example through the remainder of our discussion.

Before presenting the algorithms, we first briefly describe the notion of a *rowset*, an important underlying concept used throughout this analysis. A rowset is simply a representation of selected tuples from a table. Evaluation of a star-join query consists of two phases: creating access structures to identify which tuples are to be retrieved to answer the query, and retrieving the actual data for the selected tuples. A rowset is the access structure used in the first phase. Two approaches to representing a rowset would be to represent it as a list of *row identifiers* (RIDs) or a bit vector [98]. In a RID-list representation, a rowset can be thought of as a list structure containing a set of RIDs for selected tuples, and so the rowset cardinality is the number of selected tuples. In a bit vector representation, a rowset is a vector of bits (having cardinality of the table itself), where bits are set only for selected tuples.

The size of a rowset R, in either form, can easily be computed. For an RID-list representation, the size of the rowset is governed by the number of rows in the set, $|R|$. In this case, the minimum size of the rowset representation is $r \times |R|$, where r is the size of an RID. For a bit vector representation, the size of a rowset is governed by the number of records present in the table. It is given by $\left\lceil \frac{|T|}{8} \right\rceil \approx \frac{|T|}{8}$, where we divide by 8 to convert this value from bytes to bits. Thus, from the point of view of storage requirements, an RID-list representation is better than a bit vector if the following condition holds.

$$r \times |R| < \frac{|T|}{8} \Longleftrightarrow \frac{|R|}{|T|} < \frac{1}{8r} \tag{1}$$

In other words, a RID-list is only smaller when the *selectivity* ($\frac{|R|}{|T|}$) of the rowset is less than $\frac{1}{8r}$. In the example presented in section 3.2, where r is 6 bytes, a RID-list representation of a rowset would only be better if this rowset corresponds to less than 2% of the number of records in the underlying table. In decision support environments, many queries access significant portions of the underlying database [28]. In addition, many operations on bitmaps are much faster than on RID lists[98]. For these reasons, in this research, we assume that the rowsets used in evaluating a selection predicate are implemented as bit vectors.

Now we turn our attention to analyzing how a star-join query is evaluated in a data warehouse.

Essentially, star-join query is evaluated in two phases: the range selection phase and the join phase. In the range selection phase, the selection predicates (P_σ) are applied individually to each table that participates in the join. This results in a set of rowsets that indicate which tuples from each table are candidates for inclusion in the join result. In the join phase, the rowsets are used in conjunction with index structures to retrieve the data for tuples appearing in the join result.

To illustrate this approach, consider again our sample query that was presented earlier in this section. The set of dimension tables participating in the join is $\mathcal{D} = \{\texttt{TIME}, \texttt{CUSTOMER}, \texttt{SUPPLIER}\}$, the set of dimension table columns that contribute to the join result is $\mathcal{C_D} = \{\texttt{SUPPLIER.Name}\}$, and the set of fact-table columns that appear in the result is $\mathcal{C}_F = \{\texttt{ExtPrice}\}$.

To answer this query, we would begin the range selection phase by first applying the predicates `T.Year BETWEEN 1996 AND 1998`, `U.Nation='United States'` and `C.Nation='United States'` to the corresponding dimension tables (i.e., `TIME`, `CUSTOMER` and `SUPPLIER`). These selections would result in a set of rowsets, $\mathcal{R}$, one for each of the dimension tables involved (i.e., $\mathcal{R} = \{R_{\texttt{TIME}}, R_{\texttt{CUSTOMER}}, R_{\texttt{SUPPLIER}}\}$). Since no predicates were applied on the fact table, F, the corresponding rowset, R_F, corresponds to all tuples of F.

Once the range selection phase completes, the join phase would commence. We now discuss the execution of these two phases separately. Once we have completed describing the two phases, we will present our analysis which computes the costs of performing each phase.

3.3.1 The Range Selection Phase Using DataIndexes

In this phase, rowsets are computed based on the restriction (selection) criteria applied in the query under consideration. Performing this phase using BDIs and JDIs is quite simple.

To evaluate a selection using a BDI, it is necessary to scan the entire BDI and evaluate the selection predicate on each value in the BDI. For example, to evaluate the predicate `T.Year BETWEEN 1996 AND 1998`, it would be necessary to scan the `T.Year` BDI and generate a rowset where a set bit corresponds to an ordinal position such that the record at this position in the BDI satisfies the predicate.

A JDI is fundamentally different from a BDI in that none of the search-key values are present in the index. Rather, the RIDs corresponding to foreign records that hold these values are stored in the index. So, evaluating a predicate-based selection operation on a JDI cannot be done by only accessing the JDI. Rather, the foreign column is first scanned, and a rowset of all matching entries is generated and kept in memory. The JDI is then scanned and a second rowset is constructed by determining which entries in the JDI are present in the first rowset.

For instance, consider the TPC-D schema shown in Figure 6. Assume a JDI exists on `SALES.SuppKey` and a BDI exists on `SUPPLIER.Nation`. To evaluate the predicate "`SUPPLIER.Nation = 'United States'` " requires first scanning the `SUPPLIER.Nation` BDI and determining which values have `'United States'` in the `Nation` field. A rowset (having cardinality equal to that of the `SUPPLIER` table) indicating which entries in the `SUPPLIER` table meet this condition is then created and kept in memory. Next the JDI on `SALES.SuppKey` is scanned and compared to the first rowset to determine which entries in the `SALES` fact table meet the predicate condition. The result of this comparison is a second rowset (having cardinality equal to that of the `SALES` table) indicating the requested `SALES` records. Having described the range selection phase, we now turn our attention to describing the *join phase* evaluation of a star-join query.

3.3.2 The Join Phase

A star-join can be evaluated in a variety of ways using DataIndexes. We propose two such approaches. Each one should be used depending on the amount of available memory. We call the first approach *Star-Join with Large memory* (SJL). It is the more efficient of the two in terms of response time, but may require significant amounts of memory in certain cases. The second one is somewhat less efficient but has negligible memory requirements. We refer to it as *Star-Join with Small memory* (SJS).

3.3.2.1 SJL algorithm

The basic idea behind SJL is to perform a star join with a single pass over each table participating in the join. Clearly the performance afforded by such an algorithm would be difficult to improve upon. The SJL algorithm is shown in Algorithm 1.

To illustrate this approach, consider again our sample query that was presented earlier in this section. The set of dimension tables participating in the join is $\mathcal{D} = \{\texttt{TIME}, \texttt{CUSTOMER}, \texttt{SUPPLIER}\}$, the set of dimension table columns that contribute to the join result is $\mathcal{C}_{\mathcal{D}} = \{\texttt{SUPPLIER.Name}\}$, and the set of fact-table columns that appear in the result is $\mathcal{C}_F = \{\texttt{ExtPrice}\}$.

Now assume that the range selection phase is complete for this query, using methods outlined in section 3.3.1. The output of this phase is a set of rowsets, $\mathcal{R}$, one for each of the dimension tables involved (i.e., $\mathcal{R} = \{R_{\texttt{TIME}}, R_{\texttt{CUSTOMER}}, R_{\texttt{SUPPLIER}}\}$). Since no predicates were applied on the fact table, F, the corresponding rowset, R_F, corresponds to all tuples of F.

The SJL algorithm would then execute the join phase, beginning by loading all blocks of the dimensional BDIs where it is known that some record of interest occurs that appear in the join result (i.e., all columns in $\mathcal{C}_{\mathcal{D}}$). This is done by steps 1 through 4 in Alg. 1, by scanning rowset $R_{\texttt{SUPPLIER}}$. The result of this operation is

Algorithm 1 SJL (Star Join with Large memory)

```
Note: Needs enough memory to hold all dimension BDIs used in the join result.
Input:
    $\mathcal{D}$: set of dimension tables involved in the join.
    $\mathcal{C}_\mathcal{D}$: set of dimension table columns that contribute to the join result.
    $\mathcal{C}_F$: set of fact-table columns that contribute to the join result.
    $\mathcal{R}$: set of rowsets, one for each table in $\mathcal{D}$ and one for the fact table $F$ ($\mathcal{R} = \{R_1, \dots R_{|\mathcal{D}|}, R_F\}$). These are computed through the range selection phase, before SJL starts. Note that $R_1, \dots R_{|\mathcal{D}|}$ are already loaded into memory, whereas $R_F$ is not.
1: for each column $c_i \in \mathcal{C}_\mathcal{D}$ do
2:   for each row $r \in R_i$ do
3:     if the block of BDI on $c_i$ where $r$ is located is not loaded then
4:       Load this block into memory array $a_{c_i}$ and pin in memory
5: for each row $r \in R_F$ do
6:   for each JDI $j$ on a table of $\mathcal{D}$ do
7:     if $j(r) \notin R_j$ then
8:       goto 5
9:   for each $c_i \in \mathcal{C}_\mathcal{D}$ do
10:    $s[c_i] \leftarrow a_{c_i}(j_{c_i}(r))$
11:  for each $c_j \in \mathcal{C}_F$ do
12:    $s[c_j] \leftarrow r[c_j]$
13:  Output $s$.
```

that the appropriate blocks of the SUPPLIER.Name BDI would be in memory, along with the three rowsets generated during predicate selection (i.e., $R_{\texttt{TIME}}$, $R_{\texttt{CUSTOMER}}$, and $R_{\texttt{SUPPLIER}}$).

After this, SJL would begin scanning the appropriate SALES JDIs to determine which SALES records should appear in the join result (steps 5-8). This would proceed as follows. For each record in the fact table, the JDIs linking F to elements of $\mathcal{D}$ are examined (step 6). We will denote these JDIs as $j_{\texttt{TIME}}$, $j_{\texttt{CUSTOMER}}$ and $j_{\texttt{SUPPLIER}}$, corresponding to the three elements of $\mathcal{D}$. SJL uses these JDIs to "look up" matching entries in the corresponding rowsets (step 7). To illustrate, consider two successive fact-table rows, r_1 and r_2. SJL would first examine the entry corresponding to r_1 in $j_{\texttt{TIME}}$ (step 6). This entry, noted $j_{\texttt{TIME}}(r_1)$, is an RID onto the TIME dimension table and it can thus be used to access the corresponding bit in $R_{\texttt{TIME}}$ through a simple array look-up (step 7). Assume that this bit is cleared (i.e., set to 0); SJL would then simply skip r_1 and examine the next record (step 8). This next record, r_2, would undergo a similar set of operations. First of all, the corresponding entry in $j_{\texttt{TIME}}$ (step 6) (i.e., $j_{\texttt{TIME}}(r_2)$) would be checked against $R_{\texttt{TIME}}$ (step 7); assuming that r_2 corresponds to a sales in the years 1996 to 1998, then the next JDI, i.e., $j_{\texttt{CUSTOMER}}$ would be checked (step 6). If the corresponding bit in $R_{\texttt{CUSTOMER}}$ is set to 1 (step 7), then the last JDI (i.e., $j_{\texttt{SUPPLIER}}$) would be checked as well (step 6). Assuming that the corresponding bit in $R_{\texttt{SUPPLIER}}$ is also set to 1 (step 7), then this would indicate that r_2 does indeed appear in the join result.

Once a fact-table row r has been identified as contributing to the join result, SJL builds the corresponding join record (steps 9-12) prior to output (step 13). To do this, the corresponding entry in the in-memory BDI for SUPPLIER.Name is accessed and used to construct the output record s (step 9-10). To access the correct entry, the RID of the SUPPLIER record referenced by r is simply obtained from $j_{\texttt{SUPPLIER}}$ (step 10). This RID is mapped to an ordinal position and used to access the in-memory

BDI on SUPPLIER.Name. Since this BDI is represented as an array ($a_{\texttt{SUPPLIER.Name}}$), this step simply consists of a lookup into the array, based on the ordinal position in SUPPLIER of $j_{\texttt{SUPPLIER}}(r)$ (step 10).

Finally, the attribute values corresponding to fact-table columns are loaded from disk to complete the output record (step 11-12). In our example, the only such column is ExtPrice. The appropriate page from the ExtPrice BDI would then simply be loaded from disk and the corresponding attribute (i.e., r[ExtPrice]) would be used to finish constructing output record s (step 12).

3.3.2.2 The SJS algorithm

Recall that in SJL (steps 1-4 of Algorithm 1), the dimension table BDI's for columns appearing in the join result are loaded and pinned in memory. Thus, SJL assumes that the memory is large enough to fit all relevant columns from dimension tables in memory. Clearly, in some cases, this assumption is not realistic. For instance, referring back to Figure 6, consider a database having a scale factor of 100, which is approximately 86 GB in size. Typical OLAP queries will display dimensional attributes such as CUSTOMER.Name, a 25-byte field. A single BDI on CUSTOMER.Name would require 375 MB of memory, which exceeds the main memory of many machines today. In Algorithm 2, we present the SJS algorithm for performing a star-join on a DataIndexed warehouse, when the combined size of all relevant columns from dimension tables cannot fit in memory. Like SJL, the SJS algorithm assumes that all restrictions on participating tables have been computed and the results stored in a set of rowsets, $\mathcal{R} = \{R_1, \ldots, R_{|\mathcal{D}|}, R_F\}$, with the dimensional rowsets $\{R_1, \ldots, R_{|\mathcal{D}|}\}$ loaded into memory prior to the start of the algorithm. Thus, it is assumed that enough memory exists to load all dimensional rowsets. However, it is *not* assumed that sufficient memory exists to load all dimension table BDIs. Rather, the join is performed on smaller subsets of the dimensional BDIs by loading as many blocks of these BDIs as will fit into the available memory. Temporary structures and merge techniques are required to perform these operations. Clearly the amount of available memory plays a critical role in the performance of this algorithm, which we will analyze in more detail in a later section. For now, we describe the SJS algorithm in more detail.

Algorithm 2 SJS (Star Join with Small memory)

```
Note: Has negligible memory requirements.
Input: Same as in Algorithm 1.
 1: for each JDI $j$ on a table of $\mathcal{D}$ do
 2:   for each row $r \in R_F$ do
 3:     if $j(r) \notin R_j$ then
 4:       $R_F \leftarrow R_F - \{r\}$ /* turn corresponding bit off */
 5: for each JDI $j_t$ on a table $t \in \mathcal{D}$ do
 6:   for each row $r \in R_F$ do
 7:     write $j_t(r)$ to temporary JDI $j_{t,\text{temp}}$ on disk
 8: for each BDI $b_i$ on a column $c_i \in \mathcal{C}_\mathcal{D}$ do
 9:   Create output BDI $b_i(\text{out})$ on disk
10:   $k \leftarrow 1$
11:   while $\exists$ unloaded blocks of $b_i$ do
12:     Load as many blocks of $b_i$ as possible into in-memory array $a_i$
13:     for each row $r$ in $j_{i,\text{temp}}$ do
14:       if $b_i(r) \in a_i$ then
15:         $b_i(\text{out}) \leftarrow b_i(r)$ /* write matching entry to output BDI */
16:     $k \leftarrow k + 1$
17: for each row $r \in R_F$ do
18:   for each column $c_i \in \mathcal{C}_\mathcal{D}$ do
19:     $s[c_i] \leftarrow b_i(r)$
20:   for each column $c_j \in \mathcal{C}_F$ do
21:     $s[c_j] \leftarrow r[c_j]$
22:   Output $s$
```

We describe the algorithm in four main phases. We refer to the first phase (steps 1-4) as the *fact table rowset* (R_F) *restriction phase*. This phase restricts the initial rowset on the fact table, R_F, so that it only indicates records that will appear in the join result. This is done by accessing the JDIs of the fact table corresponding to all participating dimensional tables. This process is similar to scanning the JDIs in the SJL algorithm (steps 5-7 of Algorithm 1), except that R_F is updated. We refer to the second phase (steps 5-7) as the *JDI restriction phase.* This phase restricts the JDIs to those rows of the fact table that appear in the join result. This is done by simply scanning the restricted fact table rowset created in the previous phase. The resulting restricted JDIs are stored in temporary structures on disk. These first two phases basically prepare the data for the actual join, which we describe in the next two phases.

We refer to the third phase (steps 8-16) as the *output BDI creation phase.* This phase constructs an output BDI for each dimension table column appearing in the join result (i.e., all columns in $\mathcal{C_D}$). This is done by loading into memory as many blocks of the dimension table BDI as will fit; i.e., loading some fraction of the BDI. Then for this loaded portion of the BDI, the restricted JDI is scanned to find matching BDI entries, which are then written to the output BDI. Each value is written to the output BDI in the order corresponding to the restricted JDI, so the output BDI will have the same cardinality as the restricted JDI. The JDI and output BDI are processed sequentially, one block at a time. This processing is repeated as many times as necessary to load all dimension table BDIs. The same processing is done for all columns in $\mathcal{C_D}$.

Referring back to our example query, recall that $\mathcal{C_D} = \{$SUPPLIER.Name$\}$. Suppose the total memory required for the entire SUPPLIER.Name BDI is 250 MB, yet only 64 MB of memory is available. In other words, we assume here that 64 MB of memory is available after accounting for the memory already occupied by the JDI and output BDI blocks that are currently loaded for this phase. Clearly we cannot load all dimensional BDIs at once, but rather, can load up to 64 MB at a time. If we assume an effective block size of 8 KB, then we can load 8,000 blocks of this BDI at once ($=(\lceil\frac{64\ MB}{8\ KB}\rceil)$), which is roughly one-fourth of the SUPPLIER.Name BDI

(=($\lceil\frac{250\ MB}{64\ MB}\rceil$)). Once we have 8,000 blocks of the SUPPLIER.Name BDI loaded, the JDI for the SUPPLIER dimension, j_{SUPPLIER}, is scanned for RIDs that point to one of the BDI entries in memory. Matching entries are then written to the output BDI. This process is repeated until all blocks for the BDI have been loaded, 4 times in our example. The result is the output BDI for SUPPLIER.Name, which contains the corresponding output values in fact table order based on the restricted JDI.

We refer to the fourth phase (steps 17-22) as the *final output merge phase.* This phase creates the final output by scanning R_F and merging the dimension table output BDIs (created in the third phase) with the fact table output BDIs (i.e., all columns $\mathcal{C_F}$). In our example, for each record in R_F, the corresponding value from the output BDI for SUPPLIER.Name is obtained along with the corresponding value from the output BDI for ExtPrice. Note the simplicity of this phase since all output BDIs are in fact table order.

Having presented the DataIndex structures and star join algorithms based on these structures, we now turn our attention to analyzing the cost to perform common OLAP queries using these structures and algorithms.

Chapter 4

Comparative Analyses of Star Join Algorithms

In this chapter, we present a set of cost analyses which compare the performance of the algorithms proposed in Chaper 3 to existing star join algorithms. We first describe the metric used in the cost analyses. We then present detailed cost expressions for our proposed algorithms as well as for existing star join algorithms. Finally, based on these cost expressions, we present an analytical comparison of our proposed algorithms with a star join algorithm that is commonly used in practice.

4.1 Cost Analysis of Proposed Star Join Algorithms

In a relational system, a query is generally first translated from its original format (e.g., SQL) into some internal format (often an extension of Select-Project-Join in relational algebra) which is then used by the query optimizer to determine a query execution plan, i.e., a sequence of data and index accesses and manipulations. This plan is then executed as a series of disk accesses – which load the relevant portions of the database to main memory – interleaved with bursts of CPU activity, when the loaded data is operated upon. Mapping functions are required to determine the specific disk block that needs to be accessed and these depend on the index structure used. The mapping operations needed for DataIndexes are similar to the ones used in other systems, which utilize bitmap vectors (in bitmapped indexes or for bitmapped rowsets). Depending on the system in use, the *logical* block numbers discussed in the previous section are translated into physical block IDs either by the operating system or by low-level routines of the storage manager of the DBMS itself. The logical block numbers allow the system to work as if the files were allocated contiguous storage when in fact they are not. In all cases the mapping operations can be implemented through a few integer operations and are thus quite fast. In fact, in most cases, we believe that the delays associated with these computations will be negligible compared to the much slower storage access times. This belief is strengthened by other studies [98, 66], which have shown that I/O related costs (disk access plus I/O related CPU costs) are several orders of magnitude more than other CPU costs relating to query processing. Based on these findings, in this research, we will focus on analyzing the index structure performance with respect to disk access. Specifically, we characterize the performance of a query as the number of data blocks, $\mathcal{N}$, that are accessed during the execution of that query.

Overall, the number of block accesses necessary to perform a star-join, $\mathcal{N}_{\text{star}}$, can be expressed in terms of the cost of creating all initial rowsets and that of joining the corresponding rows together to compute the final result. More specifically, we define $\mathcal{N}_{\texttt{ROWSET}}$ to be the number of data block accesses required to construct a rowset corresponding to a particular selection predicate. We also define $\mathcal{N}_{\texttt{JOIN}}$ to be the number of block accesses required to join these rowsets to form the final query result. Putting these two costs together gives the following expression for the cost (in terms of number of block accesses) to perform a star-join:

$$\mathcal{N}_{\text{star}} = \sum \mathcal{N}_{\texttt{ROWSET}} + \mathcal{N}_{\texttt{JOIN}} \tag{2}$$

where the summation of $\mathcal{N}_{\texttt{ROWSET}}$ is performed over all selection predicates. Following this model, we now present an analysis of the cost of performing rowset constructions based on the DataIndexes and then present two separate analyses of the cost of performing the actual join based upon the two different algorithms outlined in the previous section.

First, we summarize in table 5 the notation used throughout this analysis. Note that some of the notation in this table has not yet been used.

Notation	Description
B	Effective Size, in bytes, of a data block
π	Size, in bytes, of a pointer to a data block
r	Size, in bytes, of an RID
$\|T\|$	Number of records present in table T
V	Number of distinct values present in the column being indexed
ς_T	Selectivity factor on table T ($0 \leq \varsigma_T \leq 1$)
c	Distinctness factor of range selection ($0 \leq c \leq 1$)
V_{range}	Number of distinct search-key values referenced by a particular range selection (i.e., number of all such v_k values present in the table such that $k_1 \leq v_k \leq k_2$). Note that $V_{\text{range}} = \varsigma_T\|T\|c$
$w(C)$	Width, in bytes, of a particular column C
$w(T)$	Width, in bytes, of a table T
K	Number of search key values per node of B^+-tree
P	Order of B^+-tree , i.e., $P = K + 1$
f	Compression factor such that $0 < f \leq 1$ where $f = 1$ indicates no compression
$\mathcal{D}$	Set of dimension tables involved in a join
$C_{\mathcal{D}}$	Set of dimension table columns that contribute to the join result
C_F	Set of fact-table columns that contribute to the join result
R_i	Rowset corresponding to dimension i ($i = 1, 2, \ldots, \mathcal{D}$)
R_F	Rowset corresponding to the fact table
$\mathcal{R}$	Set of rowsets, one for each table in $\mathcal{D}$ and one for the fact table F ($\mathcal{R} = \{R_1, \ldots R_{\|\mathcal{D}\|}, R_F\}$)
M	Number of blocks allocated to input BDI

Table 5: Notation used in Analyses

4.1.1 Cost for Constructing Rowsets Using DataIndexes ($\mathcal{N}_{\text{ROWSET}}$)

We now examine the first component of query cost, $\mathcal{N}_{\text{ROWSET}}$, for BDIs and JDIs. Most OLAP selection operations will consist of range predicates, i.e., having the form $k_1 \leq C \leq k_2$, where k_1 and k_2 are (possibly equal) constants and C is the column being inspected. Our analysis can easily be extended to more complex predicates.

First, to evaluate a selection using a BDI, it is necessary to scan the entire list and evaluate the selection predicate on each value in the list. Hence, the cost of evaluating a selection on a column C indexed by a BDI is simply the number of block accesses required to scan the index:

$$\mathcal{N}_{\text{ROWSET}}(\text{BDI}) = \left\lceil \frac{|T| \times w(C)}{B} \right\rceil \tag{3}$$

where $|T|$ represents the number of records in the table, $w(C)$ represents the width of the column being indexed, and B is the effective size of a data block in bytes.

To simplify the analysis, we drop the ceiling ($\lceil \cdot \rceil$) function and approximate the cost of building a rowset using a BDI to be:

$$\mathcal{N}_{\text{ROWSET}}(\text{BDI}) \approx \frac{|T| \times w(C)}{B} . \tag{4}$$

Second, as mentioned previously, a JDI is fundamentally different from a BDI in that none of the search-key values are present in the index. Rather, the RIDs corresponding to foreign records that hold these values are stored in the index. So, evaluating a predicate-based selection operation on a JDI cannot be done by only accessing the JDI. Rather, the foreign column is first scanned, and a rowset of all matching entries is generated and kept in memory. The JDI is then scanned and a second rowset is constructed by determining which entries in the JDI are present in the first rowset.

The cost to construct a rowset for the SALES fact table using this method yields the following expression, where the first term corresponds to scanning the BDI and the second term corresponds to scanning the JDI:

$$\mathcal{N}_{\text{ROWSET}}(\text{JDI}) = \left\lceil \frac{V\, w(C)}{B} \right\rceil + \left\lceil \frac{|T| r}{B} \right\rceil \qquad (5)$$

Here V represents the number of distinct values present in the column being indexed. It is assumed that the foreign column is a primary key of the dimension table and is stored as a BDI. Thus the cardinality of the referenced BDI is V. Note that V is usually much smaller than $|T|$. In the second term, r represents the size of a RID in bytes. Applying the same simplifications as before results in the following expression for the cost to construct a rowset using a JDI:

$$\mathcal{N}_{\text{ROWSET}}(\text{JDI}) \approx \frac{V\, w(C) + |T| r}{B} \qquad (6)$$

4.1.2 Cost for Joining Tables ($\mathcal{N}_{\text{JOIN}}$)

In this section we examine the cost to perform a join using both the SJL algorithm, $\mathcal{N}_{\text{JOIN}}(\text{SJL})$, and the SJS algorithm, $\mathcal{N}_{\text{JOIN}}(\text{SJS})$. In this analysis we assume that disk fragmentation is neglible. Unlike transactional processing systems, in a data warehouse, the emphasis is on querying rather than updating. Updates in a data warehouse typically occur in bulk, so records will be packed. It is therefore reasonable to assume that the degree of fragmentation will be insignificant and so we assume packed records throughout this analysis.

4.1.2.1 Cost of the SJL Algorithm

It should be clear from the discussion in Section 3.3.2.1 that SJL performs a single scan of the central SALES fact table, and that only a limited number of columns is ever examined. During this scan, only pages that correspond to records in R_F are actually considered (step 5 of Algorithm 1), and often, only a subset of the corresponding JDI pages will need to be loaded and examined (steps 6-8 of Algorithm 1). Indeed, a query optimizer should determine the order in which these JDIs should be examined, so as to minimize the number of page accesses. A simple rule of thumb for this type of optimization would be to select the JDI whose corresponding rowset has the smallest selectivity. Finally, once a record is known to participate in the join, only a subset of the columns from the different tables is ever accessed (steps 9-12 of Algorithm 1).

This simple approach allows for very efficient star-join evaluation. Indeed, the cost of a query on a fact table F and a set of dimension tables $\mathcal{D}$ can be expressed as follows:

$$\mathcal{N}_{\text{JOIN}}(\text{SJL}) = \mathcal{N}_{\mathcal{D}}(\text{BDI}) + \mathcal{N}_F(\text{JDI}) + \mathcal{N}_F(\text{BDI}) \tag{7}$$

where $\mathcal{N}_{\mathcal{D}}(\text{BDI})$represents the cost of retrieving all blocks containing relevant records from dimensional BDIs, $\mathcal{N}_F(\text{JDI})$, the cost of scanning each of the relevant JDIs from the fact table, and $\mathcal{N}_F(\text{BDI})$, the cost of scanning all relevant records from fact table BDIs. We now derive expressions for each of these terms.

The cost to retrieve the relevant blocks for dimensional BDIs requires scanning the dimensional rowset for each dimension table column involved in the join result. This cost can be expressed as follows:

$$\mathcal{N}_{\mathcal{D}}(\text{BDI}) = \sum_{D \in \mathcal{D}} \sum_{C \in \mathcal{C}_D} \left\lceil \frac{|D| \times w(C)}{B} \right\rceil \tag{8}$$

where $\mathcal{D}$ represents the set of dimension tables involved in the join, $|D|$ represents the cardinality of the rowset for dimension table D, and $\mathcal{C}_D$ represents the set of columns from a table D that appear in the join result.

The cost to scan the relevant JDIs from the fact table is given by:

$$\mathcal{N}_F(\text{JDI}) = |\mathcal{D}| \left\lceil \frac{r|F|}{B} \right\rceil \tag{9}$$

Here $\left\lceil \frac{r|F|}{B} \right\rceil$ represents the number of block accesses required for a particular JDI and $|\mathcal{D}|$ the number of dimension tables participating in the join. In the worst case, all foreign colums satisfy the restriction conditions and so all $|\mathcal{D}|$ JDIs must be examined.

Finally, the last part of the algorithm involves loading columns from the fact table to complete the output record. The cost to scan the relevant records from the fact table is given by:

$$\mathcal{N}_F(\text{BDI}) = \sum_{C \in \mathcal{C}_F} \min \left(\varsigma_F |F|, \left\lceil \frac{|F| \times w(C)}{B} \right\rceil \right) \tag{10}$$

Here ς_F denotes the final selectivity on the fact table (i.e., the number of records in the join is $\varsigma_F |F|$ and $0 \leq \varsigma_F \leq 1$). The cost of loading the fact table columns depends on the selectivity. This cost will be the lesser of the number of records in the join (i.e., the first term in (10) by random access), and the total number of blocks required for all relevant columns in F (i.e., the second term in (10) by sequential scan of the entire index).

By inspection, we can thus establish that the dominant factor in these equations is $|F| \times |\mathcal{D}|$, which indicates that the worst-case performance of this algorithm will be $O(|F| \times |\mathcal{D}|)$ or simply $O(|F|)$ since $|\mathcal{D}|$ is bounded by a small constant for a given star schema.

We note again that this efficient approach can only be utilized if enough memory can be allocated to the query. This memory requirement is given by

$$\mathcal{M}_{\text{JOIN}}(\text{SJL}) = 1 + |\mathcal{D}| + |\mathcal{C}_F| + \sum_{D \in \mathcal{D}} \sum_{C \in \mathcal{C}_D} \left\lceil \frac{|D| \times w(C)}{B} \right\rceil + |\mathcal{D}| \sum_{D \in \mathcal{D}} \left\lceil \frac{|D|}{8B} \right\rceil . \quad (11)$$

Here the first term corresponds to one block of memory for the fact table, the second term corresponds to $|\mathcal{D}|$ blocks for the JDIs, and the third term corresponds to $|\mathcal{C}_F|$ blocks for the fact table BDIs. Thus we assume that the algorithm proceeds by accessing the fact table rowset, each JDI, and each fact table display column one block at a time. The fourth term corresponds to the memory requirements for the dimension table BDIs, and the last term corresponds to the memory requirements for the dimension table rowsets, both of which are loaded into memory for the duration of the algorithm.

From 11, we can conclude the following:

Result 1 *The memory requirements for the SJL algorithm are independent of the size of the fact table.*

This is an interesting result because it allows us to see that the SJL algorithm often does not require much memory, and that the memory requirements do not increase as the size of the fact table increases.

To illustrate, consider again the star-join query from Section 3.3, which joins the SALES fact table with the TIME, CUSTOMER, and SUPPLIER dimension tables. Using (11), it is easily shown that a total of 52 blocks of memory are required to answer the query using SJL. While this is by no means a "monster" query, it is certainly respectable. Yet, on any given system, it would only require about 416KB of memory, regardless of the size of the fact table. This amount of memory is small enough as to be even available on low-end personal computers. In addition, the corresponding columns could be used concurrently by other queries, thereby reducing the effective memory requirements of each query. This can be done because SJL always loads the entire columns "as-is", without pre-performing selections or reordering the data. It would thus appear that the memory requirements of SJL are indeed acceptable for many OLAP-type queries.

4.1.2.2 Cost of the SJS Algorithm

We now present the cost to perform a join using the SJS approach. This cost is given by:

$$\mathcal{N}_{\text{JOIN}}(\text{SJS}) = \mathcal{N}_{R_F} + \mathcal{N}_{\text{JDI}} + \mathcal{N}_{\text{OBDI}} + \mathcal{N}_{\text{Merge}} \tag{12}$$

where $\mathcal{N}_{R_F}$represents the cost of the R_F restriction phase, (steps 1-4 of Algorithm 2), $\mathcal{N}_{\text{JDI}}$represents the cost of the JDI restriction phase (steps 5-7), $\mathcal{N}_{\text{OBDI}}$represents the cost of the output BDI creation phase (steps 8-16), and $\mathcal{N}_{\text{Merge}}$represents the cost of the final output merge phase (steps 17-22). The expressions for each of these terms is given below. The cost to restrict R_F is given by:

$$\mathcal{N}_{R_F} = 2\left\lceil \frac{|F|}{8B} \right\rceil + \mathcal{N}_F(\text{JDI}) \tag{13}$$

where $\frac{|F|}{8}$ is the size of R_F (in bits). Dividing by B gives $\left\lceil \frac{|F|}{8B} \right\rceil$, the total number of blocks required to store R_F. Since the entire rowset must be loaded and written back to disk, the first term thus represents the cost both to load and write the rowset. The second term represents the cost to load the JDIs and is given by (9).

The cost to restrict the JDIs is given by:

$$\mathcal{N}_{\text{JDI}} = \left\lceil \frac{|F|}{8B} \right\rceil + \mathcal{N}_F(\text{JDI}) + |\mathcal{D}|\mathcal{N}_R(\text{JDI}) \tag{14}$$

where the first term represents the cost to load R_F and the second term represents the cost to load the JDIs and is given by (9). The third term represents the cost to write the new restricted JDIs where $\mathcal{N}_R(\text{JDI})$ is given by $\left\lceil \frac{r|F'|}{B} \right\rceil$ and $|F'| = \varsigma_F \varsigma_D^{|\mathcal{D}|} |F|$. Here ς_F represents selectivity on the fact table and similarly, ς_D represents selectivity on dimension table D. For simplicity, it is assumed that selectivity is the same for all dimension tables. The effect of this restriction is to reduce the number of relevant tuples based on ς_F and ς_D. Hence the cardinality of each JDI is reduced to $|F'|$.

The cost to create the output BDIs can be expressed as:

$$\mathcal{N}_{\text{OBDI}} = \sum_{D \in \mathcal{D}} \sum_{C \in \mathcal{C}_D} L_C \left(\mathcal{N}_R(\text{JDI}) + \frac{\mathcal{N}_i(C)}{L_C} + \mathcal{N}_o(C) \right) \tag{15}$$

For each participating dimension table and for each column appearing in the join result, several passes may have to be made in order to scan the restricted JDI, locate the associated dimension table column value, and then write the value to the output BDI. We let L_C represent the number of passes required, which is given by $\left\lceil \frac{\mathcal{N}_i(C)}{M} \right\rceil$, where $\mathcal{N}_i(C)$ is the total number of blocks required for the input BDI and is given by $\left\lceil \frac{|D| \times w(C)}{B} \right\rceil$. M is the number of blocks allocated to the input BDI and depends on the *available* memory, which is the total memory less what is already loaded. We discuss memory requirements of SJS in more detail later in this section. $\mathcal{N}_o(C)$ represents the total number of blocks required for the output BDI and is given by $\left\lceil \frac{|F'| \times w(C)}{B} \right\rceil$. Returning to the actual processing, for each pass, all blocks of the corresponding JDI must be loaded (the first term in (15)), the number of blocks of the input BDI that fit in memory must be loaded (the second term in (15)), and all blocks of the output BDI must be accessed (the third term in (15)).

The expression in (15) can be simplified to give the following:

$$\mathcal{N}_{\text{OBDI}} = \mathcal{N}_R(\text{JDI}) \sum_{D \in \mathcal{D}} \sum_{C \in \mathcal{C}_D} L_C + \mathcal{N}_{\mathcal{D}}(\text{BDI}) + \sum_{D \in \mathcal{D}} \sum_{C \in \mathcal{C}_D} L_C \mathcal{N}_o(C) \tag{16}$$

where $\mathcal{N}_{\mathcal{D}}(\text{BDI})$is the cost to retrieve all blocks containing relevant dimension table BDIs and is given by (8).

The cost to create the final output can be expressed as:

$$\mathcal{N}_{\text{Merge}} = \left\lceil \frac{|F|}{8B} \right\rceil + \mathcal{N}_F(\text{BDI}) + \sum_{D \in \mathcal{D}} \sum_{C \in \mathcal{C}_D} L_C \mathcal{N}_\circ(C) \tag{17}$$

where the first term corresponds to the cost to load R_F, the second term corresponds to the cost to load the fact table output BDIs and is given by (10), and the third term corresponds to the cost to load the dimension table output BDIs created in the previous phase.

The minimum memory requirements for SJS are quite small and are given by:

$$\mathcal{M}_{\text{JOIN}}(\text{SJS}) = 1 + |\mathcal{C}_{\mathcal{D}}| + |\mathcal{C}_F| + \sum_{D \in \mathcal{D}} \left\lceil \frac{|D|}{8B} \right\rceil . \tag{18}$$

Since each phase in SJS has different memory requirements, (18) is based on the memory requirements for the final output merge phase, which requires the most memory of all phases. Therefore, the first term corresponds to one block of memory for R_F, the second term corresponds to one block for each of the dimension table output columns, and the third term corresponds to one block for each of the fact table output columns. Finally, the last term corresponds to the number of blocks required for the dimensional rowsets.

Having analyzed the query processing cost based on DataIndexes, we now turn our attention to different types of indexing structures. In the next section, we present an analysis of the cost to perform a star-join query using different indexing structures.

4.2 Comparative Analyses of Existing Indexing Techniques

The cost to perform a star-join query, as expressed in equation (2), is the sum of the cost to construct rowsets corresponding to all selection predicates ($\sum \mathcal{N}_{\texttt{ROWSET}}$), and the cost to join these rowsets ($\mathcal{N}_{\texttt{JOIN}}$). Following this model again, we analyze the cost of evaluating star join processing using different techniques and index structures. In the ensuing analysis we consider virtually all the state-of-the-art access structures used in data warehouses currently. For rowset selections, we use *B-Tree* indexes, Bitmap indexes, *Bit-sliced* indexes and *Projection Indexes*, while for joins we use the *Bitmap Join Index*. In each case, we first derive expressions that yield the *best case expected performance* of each approach and then compare these results to determine which approach is the most promising, and under what conditions.

4.2.1 Comparative Anaysis of $\mathcal{N}_{\texttt{ROWSET}}$

4.2.1.1 $\mathcal{N}_{\texttt{ROWSET}}$ for B$^+$-tree Index

Possibly the most common indexing scheme available is the B$^+$-tree . This structure consists of a balanced tree whose nodes occupy each a single data block. The data blocks in the leaf level make up a sorted list of the V distinct search-key values in the column being indexed (in our sample query above, this would match, for instance, the number of unique values for the `Nation` field in the `Customer` table). Attached to each one of the unique values is a list of the RIDs of the records corresponding to that value. B$^+$-trees are often implemented so as to reduce the number of tree reorganizations necessary when the underlying data is updated. This translates to an average utilization of about 69% for the nodes in the tree [34]. While this is indeed useful in transaction processing systems, this overhead is not needed in the read-mostly environment of data warehouses. In this study, thus, we assume that the tree is optimally filled (i.e., almost all nodes are full). We also assume that all values obtained in a range query are contiguous.

The cost to construct a selection predicate rowset with a B-tree can be expressed as follows:

$$\mathcal{N}_{\text{ROWSET}}(\text{B-tree}) = \mathcal{N}_{\text{descent}}(\text{B-tree}) + \mathcal{N}_{\text{leaf}}(\text{B-tree}) + \mathcal{N}_{\text{RID-list}}(\text{B-tree}) \qquad (19)$$

where $\mathcal{N}_{\text{descent}}$ is the cost of descending the tree, $\mathcal{N}_{\text{leaf}}$ is the cost of scanning the leaf-level for all matching entries, and $\mathcal{N}_{\text{RID-list}}$ is the cost of actually accessing all RID-lists. We now derive expressions for each of these components.

The cost to descend a B$^+$-tree depends on the number of levels in the tree. The number of levels is given by $\lceil \log_P V \rceil$, where P is the *order* of the tree and V represents the number of distinct values present in the column being indexed. The order of the tree is $K + 1$ where K is the number of search key values per node and K is given by $\left\lceil \frac{B}{w(C)+\pi} \right\rceil$. This expression determines the number of key values that can fit in a node given the size of each key value, pointer pair $(w(C) + \pi)$, and the effective blocksize, B. Since we do not need to include the leaf level, the cost to descend the tree is then as follows:

$$\mathcal{N}_{\text{descent}}(\text{B-tree}) = \lceil \log_P V - 1 \rceil \qquad (20)$$

The cost of scanning the leaf-level is the number of blocks accessed at the leaf level and is given by:

$$\mathcal{N}_{\text{leaf}}(\text{B-tree}) = \left\lceil \frac{V_{\text{range}}}{K} \right\rceil \qquad (21)$$

where V_{range} is the number of distinct search-key values referenced by the range selection and K is as defined previously. This expression follows from the fact that there are V_{range} distinct search-key values in the range and K key values per node. Note that in the best case, this expression evaluates to one since only a single block access is required.

The cost of accessing the RID-lists is given by the following expression:

$$\mathcal{N}_{\text{RID-list}}(\text{B-tree}) = \left\lceil \frac{r|T|}{V \times B} \right\rceil V_{\text{range}} \tag{22}$$

where r is the size of a RID in bytes, $|T|$ is the number of records present in the table, V_{range} is the number of distinct search-key values referenced by the range selection, and B and V are as defined previously. In the sample query from the previous section, V_{range} for the predicate "`T.Year BETWEEN 1996 AND 1998`" is 3, since the range covers 3 years. In deriving this expression, we assume that the distribution of distinct values is uniform. Thus the average number of RIDs per RID-list is given by $\frac{|T|}{V}$. The size of a RID-list, in bytes, is then $\frac{r|T|}{V}$. Dividing by the effective blocksize then gives the number of blocks per RID-list, $\left\lceil \frac{r|T|}{V \times B} \right\rceil$. Finally, multiplying by the number of distinct search-key values in the range results in the number of blocks required to access the RID-lists, as given in equation (22).

To simplify the analysis, we drop all ceiling ($\lceil \cdot \rceil$) functions and approximate the cost of building a rowset using a B$^+$-tree to be:

$$\mathcal{N}_{\text{ROWSET}}(\text{B-tree}) \approx \log_P V - 1 + \frac{V_{\text{range}}}{K} + \left(\frac{r|T|}{V \times B} \right) V_{\text{range}} \tag{23}$$

4.2.1.2 $\mathcal{N}_{\text{ROWSET}}$ for a Bitmap Index

A bitmapped index is identical to a conventional B-tree except that the rowsets corresponding to each unique search-key value are represented as bit vectors instead of RID lists [98]. As in the case of the B$^+$-tree , the cost of performing a range selection with a bitmapped index can be expressed as follows:

$$\mathcal{N}_{\text{ROWSET}}(\text{Bitmap}) = \mathcal{N}_{\text{descent}}(\text{Bitmap}) + \mathcal{N}_{\text{leaf}}(\text{Bitmap}) + \mathcal{N}_{\text{RID-list}}(\text{Bitmap}) \quad (24)$$

where $\mathcal{N}_{\text{descent}}$ and $\mathcal{N}_{\text{leaf}}$ are exactly the same as for the B$^+$-tree . The difference in these two structures appears in the third term, the cost to access the RID-lists, since these are stored differently in the two structures. In practice, in a bitmapped index, a certain amount of compression is typically employed in storing the bitmaps. We thus assume a compression factor f $(0 < f \leq 1)$, which is a percentage indicating the compression level ($f = 1$ indicates no compression). We can then express $\mathcal{N}_{\text{RID-list}}$ as follows:

$$\mathcal{N}_{\text{RID-list}}(\text{Bitmap}) = f \left\lceil \frac{|T|}{8\ B} \right\rceil V_{\text{range}} \quad (25)$$

where $\frac{|T|}{8}$ is the size of a bit vector in bytes, and so $\left\lceil \frac{|T|}{8\ B} \right\rceil$ represents the average number of blocks per bit vector. Multiplying this expression by V_{range} gives the total number of blocks accessed, which is then weighted by the compression factor. Applying the same simplifications as in (23), the rowset construction cost using a bitmapped index can be expressed as:

$$\mathcal{N}_{\text{ROWSET}}(\text{Bitmap}) \approx \log_P V - 1 + \frac{V_{\text{range}}}{K} + f\left(\frac{|T|}{8\ B}\right) V_{\text{range}} \quad (26)$$

From equations 23 and 26, there appears to be a tradeoff between B$^+$-tree and bitmapped indexes. However, in practice, bitmapped indexes almost always require less storage than B$^+$-tree as significant compression can usually be achieved on the bitmaps. We now provide an illustrative example. A simple compression technique used in bitmapped indexes [98] is to represent the rowsets as bitmaps *only when the bitmap representation is smaller than a RID list representation.* It is easily seen that a bitmapped index constructed according to this method *can never require more* *storage than a B^+-tree* . Of course, this compression technique is quite simple and more effective compression mechanisms can be used. Thus we conclude the following result:

Result 2 *The performance of a bitmapped index is never worse than that of a conventional B-tree index.*

Because of this result, in the remainder of this analysis, we do not consider the B^+-tree but rather concentrate on the bitmapped index.

4.2.1.3 $\mathcal{N}_{\text{ROWSET}}$ for a Projection Index

A projection index corresponds to a mirror copy of the column being indexed [98]. Like a single column BDI, to evaluate a selection using a projection index, it is necessary to scan the entire list and evaluate the selection predicate on each value in the list.

4.2.1.4 $\mathcal{N}_{\text{ROWSET}}$ for a Bit-sliced Index

Like the projection index and BDI, the bit-sliced index[1] also scans the entire index. In addition, each slice must be accessed in turn, which requires that a small header that points to each of the bit-slices will need to be accessed. Since the cost to access each slice is usually quite small, we can disregard this cost. Thus the actual cost incurred with these two techniques can be expressed as:

$$\mathcal{N}_{\text{ROWSET}}(\text{Bit-sliced}) = \mathcal{N}_{\text{ROWSET}}(\text{Projection}) = \left\lceil \frac{|T| \times w(C)}{B} \right\rceil \tag{27}$$

[1][98] gives an efficient algorithm for performing range queries on bit-sliced indexes. This algorithm uses multiple bit vectors to compute the final rowset. These intermediary bit vectors are generated by scanning each bit slice in the index.

Applying the usual simplifications, the cost of evaluating a selection on a column C indexed by either of the above two methods is simply given by:

$$\mathcal{N}_{\text{ROWSET}}(\text{Bit-sliced}) \approx \mathcal{N}_{\text{ROWSET}}(\text{Projection}) \approx \frac{|T| \times w(C)}{B} \tag{28}$$

Note that equations (27) and (28) are equivalent to (3) and (4), respectively. Thus we conclude that the cost to construct a rowset using either a projection index, bit-sliced index, or BDI are essentially the same.

4.2.1.5 Cost comparisons

Having determined the best-case performance of rowset evaluation with different techniques, we can now compare these performances to understand the conditions under which a particular scheme performs the best. To lend some structure to these comparisons, we classify the indexing mechanisms into two classes: (a) *Indexing techniques based on ordinal positions*, which include projection indexes, bit-sliced indexes, BDIs and JDIs, and (b) *Tree Based Indexing Techniques*, which include B$^+$-trees and Bitmapped indexes.

Comparison of Indexing Techniques based on Ordinal Positions

From expressions (4) and (28), it can be seen that the performances of projection indexes, BDIs and bit-sliced indexes is equivalent. We thus turn our attention to comparing BDIs and JDIs. Using (4) and (6), we can determine that the performance of a BDI is better than that of a JDI if

$$\frac{r}{w(C)} \geq 1 - \frac{V}{|T|}\,. \tag{29}$$

This result provides an easy decision guideline for the physical design of a table in a data warehouse. The right-hand side of this inequality can never be greater than 1. Thus, we immediately note the following result.

Result 3 *Contrary to popular belief, a BDI does not always perform better than a JDI. In fact, a JDI performs better if $r < w(C)$, that is, if the size of a RID is smaller than the width of the column.*

More precisely, the higher the ratio $\frac{V}{|T|}$ of the column of interest, the smaller the right hand side of the above equation and hence the larger the value of $w(C)$ for which BDI is preferable (for a given r).

Recall that we proposed the JDI only for foreign key columns. An alternative approach using a JDI would consist of a BDI containing each of the V distinct values taken by the column and an associated JDI onto that BDI, to map these V values to records in the table. To illustrate, let us consider the `SALES.ShipMode` column from the example in section 3.2; the TPC-D benchmark indicates that this can take one of 9 different values. Equation 29 clearly indicates that `ShipMode` would be better represented as a JDI with an associated look-up BDI (e.g., `ShipMode_Type`) than as a simple BDI; indeed, (29) indicates $\frac{r}{w(C)} = 0.6 \leq 1 - \frac{V}{|T|} \approx 1$. In this case, the JDI/BDI combination would require only 4503 blocks, whereas a simple BDI would need 7500 blocks of storage. Hence, it may be more efficient to implement a wide, repeating column as a JDI with an associated look-up BDI than as a simple BDI.

Comparison of Tree-Based Techniques and Ordinal-Position Based Techniques

Since we have shown already that a bitmapped index can never cost more than a B-tree index, we begin by comparing bitmapped indexes with BDIs. By comparing the last term of (26) (the first two terms are usually quite small and can thus be disregarded to simplify the analysis) and (4), it is easily seen that a BDI will perform better if:

$$V_{\text{range}} \geq \frac{8\ w(C)}{f} \ . \tag{30}$$

This simply states that, for a given compression factor, a BDI performs better than a bitmapped index when the selection range of a query is 'large'. For example, suppose f is 0.2 and the indexed column is `CUSTOMER.AcctBal` from the star schema of Figure 6, which has a width $w(C)$ of 8 bytes. This column appears to be a reasonable candidate for indexing since it is likely that account balances would be queried frequently. The above relationship indicates that a BDI will perform better if the number of distinct values in the selection range (i.e., V_{range}) is at least 320. Note that even though we claim that a BDI performs better for 'large' values of V_{range}, 320 is not really that large when compared to the cardinality of the `CUSTOMER` table, 150,000. Larger values of f would imply that a BDI is preferable for even smaller values of V_{range}. Thus we can conclude the following:

Result 4 *A BDI outperforms a bitmapped index if the number of search-key values in the predicate range is greater than the ratio of the width of the column to the compression factor.*

Note that BDIs, like bitmapped indexes, can also be compressed, which would improve their performance. However, in this analysis, we compare uncompressed BDIs to compressed bitmapped indexes. In other words, we compare the worst case BDI to the best case bitmapped index.

A similar analysis can be performed to compare the performance of bitmapped indexes and JDIs. From (26) and (6), it is easily seen that a JDI will perform better if:

$$V_{\text{range}} \geq 8\left(\frac{V\, w(C) + r|T|}{f|T|}\right) . \tag{31}$$

Similar to the results obtained for the BDI, we conclude the following:

Result 5 *A JDI outperforms a bitmapped index if the number of search-key values in the predicate range is relatively large.*

These results conclude our study of the rowset construction performance of the different indexing schemes under study. The results of this analysis can be summarized as follows. Of the indexes that rely on the ordinal position of records, the best performing ones are the JDI or the BDI (or the projection index). The cutoff point between the two occurs when $\frac{r}{w(C)} = 1 - \frac{V}{|T|}$. However, the bitmapped index sometimes performs better than each one of these approaches, but this would only occur when V_{range} is relatively small. We summarize the above results in table 6, where **BI** denotes the bitmapped index.

	Better than BI if	**Better than BDI if**	**Better than JDI if**
BI	-	$V_{\text{range}} < \frac{8\, w(C)}{f}$	$V_{\text{range}} < 8\left(\frac{V\, w(C)+r\|T\|}{f\|T\|}\right)$
BDI	$V_{\text{range}} \geq \frac{8\, w(C)}{f}$	-	$\frac{r}{w(C)} \geq 1 - \frac{V}{\|T\|}$
JDI	$V_{\text{range}} \geq 8\left(\frac{V\, w(C)+r\|T\|}{f\|T\|}\right)$	$\frac{r}{w(C)} < 1 - \frac{V}{\|T\|}$	-

Table 6: Comparison of the rowset-construction performance of the different indexes under study

Having examined the rowset construction performance of different access schemes, we now turn our attention to the second part of the cost of performing star joins, namely $\mathcal{N}_{\text{JOIN}}$, the cost of performing a join on restricted tables.

4.2.2 Comparative Analysis of $\mathcal{N}_{\text{JOIN}}$

Bitmapped indexes are used in Oracle 8 and BJIs in the Informix Universal Server. Though we would have liked to compare the performance of our algorithms with that of Red Brick's STARjoin approach, we were unable to obtain enough information to model STARjoins with any level of detail. However, as will become clear later on in this section, the high-performance of our algorithms is strongly tied to the fact that DataIndexes do not store table rows contiguously. The Red Brick system, however, relies on conventional storage approaches; hence we believe that our proposed algorithms also outperform the STARjoin approach for the majority of OLAP-type queries. It would, however, be interesting to verify -or refute- this belief.

Also, we note that some of the indexes analyzed in section 4.2.1.1 (namely projection indexes and bit-sliced indexes) seem to provide little help in computing joins. Our analysis in section 4.2.1.1 also indicates that bitmapped indexes will never perform worse than B^+-trees . We thus do not investigate the performance of star joins using projection indexes, bit-sliced indexes or B^+-trees . Instead, we concentrate on the approaches based on bitmapped indexes, BJIs and DataIndexes.

4.2.2.1 Bitmapped-Join Indexes

Based on BJIs, a star join algorithm proceeds roughly as follows. The dimensional rowsets (i.e., $R_1, \ldots R_{|\mathcal{D}|}$) computed during the predicate selection phase are used to determine the set of matching records in the fact table with the corresponding BJI. In other words, for a dimension table, D, the leaf-level of the corresponding BJI is scanned, and the rowsets associated with rows that appear in rowset R_D are loaded and bitwise OR'ed together. When these operations have been applied to all participating dimension tables, the result is the *join rowset* R_{JOIN} which indicates which records of the fact table appear in the join result. R_{JOIN} can then be used in one of two ways, depending on the amount of available memory.

In the first approach, all relevant columns and rows from the dimension tables (including the primary key column) are extracted from the dimension tables and pinned in memory. The algorithm then proceeds similarly to the SJL algorithm we proposed: the fact table F is scanned in R_{JOIN} order, and the result rows are constructed from the in-memory structures, then output.

Hence, we can compute that the overall *best case* cost of performing the join with bitmapped indexes, given that enough memory is available, and assuming that all rowsets are packed (for reasons stated previously), is:

$$\mathcal{N}_{\text{JOIN}}(\text{BJI}) = \mathcal{N}_{R_{\text{JOIN}}} + \mathcal{N}_{\text{Dim}} + \mathcal{N}_{F\text{-scan}} \;, \tag{32}$$

where $\mathcal{N}_{R_{\text{JOIN}}}$represents the cost of forming the join-rowset (R_{JOIN}), $\mathcal{N}_{\text{Dim}}$the cost of loading all dimensional tuples of interest, and $\mathcal{N}_{F\text{-scan}}$the cost of scanning the fact-table itself. We now derive expressions for each of these components. To form R_{JOIN} requires descending the tree, scanning the leaf level, and then loading the blocks having rowsets that appear in the join. Thus the cost to form R_{JOIN} , can be expressed as follows:

$$\mathcal{N}_{R_{\text{JOIN}}} = \sum_{D \in \mathcal{D}} \left(\lceil \log_{P_D} V_D - 1 \rceil + \min\left(\varsigma_D |D|, \left\lceil \frac{|D|}{K_D} \right\rceil \right) + f\left(\left\lceil \frac{|F|}{8\,B} \right\rceil \varsigma_D |D| \right) \right) \tag{33}$$

where the first term corresponds to the cost to descend the tree, the second term corresponds to the cost to scan the leaf level, and the third term corresponds to the cost to load the relevant blocks of rowsets. In the first term, P_D represents the order of the tree for dimension table D and V_D represents the number of distinct search key values in D. Both of these terms are as defined in section 4.2.1.1, with the exception that the width of each column, pointer pair is different. Specifically, an RID is contained in each node rather than a column value. Thus, the expression for P_D is then $K_D + 1$, where K_D represents the number of search keys per node in the index for D and $K_D = \lceil \frac{B}{r+\pi} \rceil$. In the second term, $|D|$ represents the cardinality of dimension table D and ς_D represents the selectivity on D. In the best case, only $\varsigma_D|D|$ blocks must be accessed at the leaf level, whereas in the worst case, all blocks must be accessed.

The cost to load dimensional tuples, $\mathcal{N}_{\text{Dim}}$, is given by:

$$\mathcal{N}_{\text{Dim}} = \sum_{T \in \mathcal{D}} \left\lceil \frac{|T| w(T)}{B} \right\rceil \tag{34}$$

where $w(T)$ represents the width of dimension table T. Note that in this case the entire tuple (all columns) must be loaded.

Finally, the cost to scan the fact table, $\mathcal{N}_{F\text{-scan}}$, is given by:

$$\mathcal{N}_{F\text{-scan}} = \min\left(\varsigma_F |F|, \left\lceil \frac{|F| w(F)}{B} \right\rceil\right) \tag{35}$$

where $w(F)$ represents the width of the fact table. This cost depends on the selectivity of F. In the best case, only $\varsigma_F|F|$ blocks must be accessed, whereas in the worst case, all blocks must be accessed.

The total memory requirements for this first approach can be expressed as:

$$\mathcal{M}_{\text{JOIN}}(\text{BJI}) = 1 + |\mathcal{D}| + \sum_{D \in \mathcal{D}} \left\lceil \frac{|D| w(D)}{B} \right\rceil \tag{36}$$

where the first term corresponds to a block of memory for the fact table, the second term corresponds to a block of memory for each dimension table, and the third term corresponds to the memory required for pinning the relevant dimension tables in memory.

The second approach for evaluating star-joins with BJIs is based on the pairwise, hash-join technique. It should be applied when there is not enough memory to load all necessary dimensional columns into memory. Once R_{JOIN} has been determined, the relevant values from the different tables are extracted from the source tables and stored in temporary files. These temporary files are then joined pair-wise until the final join result is computed, thereby requiring $|\mathcal{D}|$ individual two-way joins. Since it is well known [98] that pairwise joins do not perform well in a data warehouse environment, we do not include the cost analysis of this approach.

4.2.2.2 Bitmapped Indexes

Bitmapped indexes may also be used to perform a star-join similarly to BJIs. However, since bitmapped indexes are single-table structures, more operations are required during a join. Specifically, while a BJI contains RIDS and thus allows direct access to a particular dimension table, a bitmapped index contains values and therefore requires accessing the primary key values from the participating dimensional table tuples. These tuples are then used to scan the bitmapped indexes on the corresponding fact table columns, resulting in additional accesses in creating the join rowset R_{JOIN}. These additional accesses result in approximately the same number of block accesses as a pairwise join between the fact table and each of the dimension tables. In addition, the size of the tree structure of each index might be slightly different from those used in BJIs. For instance, the values in bitmapped indexes may vary in size, while the RIDS in BJIs are typically rather small and constant in size. Based on these differences, the following result can easily be shown:

Result 6 *Bitmapped join indexes outperform bitmapped indexes for evaluating star joins.*

4.2.2.3 Cost comparison of Bitmapped-Join and DataIndexes

We now compare the performance of DataIndexes and bitmapped join indexes under packed conditions. To do so, we compare the worst-case performance expressions for SJL to the best case expressions for bitmapped join indexes. By comparing (and making the usual simplifications) to (7) and (32), it can be shown that SJL can outperform BJI and other approaches if the following condition holds true:

$$\frac{\sum_{D \in \mathcal{D}} \varsigma_D |D|}{|\mathcal{D}|} \geq \frac{8r}{f}. \tag{37}$$

To derive this condition, we have assumed that all columns from the fact table appear in the output (i.e., $\sum_{C \in \mathcal{C}_{\mathcal{F}}} w(C) = w(F)$) and that all dimension table columns for participating dimension tables also appear in the output (i.e., $\sum_{C \in \mathcal{C}_D} w(C) = w(D)$). These assumptions are strongly biased towards traditional approaches. Clearly, the fewer the number of output columns, the better DataIndexes will perform, as only the relevant columns will need to fetched, unlike traditional approaches where all columns will be fetched, regardless of the desired output. Thus, by making the assumption that all fact table columns are needed for output, we nullify a strong advantage of DataIndexes. Even then, SJL outperforms the other approaches in a number of cases. Indeed the above condition can be understood as follows:

Result 7 *SJL outperforms other star-join approaches if the average selectivity on the dimension tables is greater than the ratio of the RID size to the compression factor.*

In other words, SJL outperforms other star-join approaches if the average number of tuples accessed from each dimension table is large. Referring again to the star schema of Figure 6, if we assume a RID size r of 6 bytes and a compression factor f of 0.2, then at least 240 tuples must be selected, on average, from a dimension table for SJL to outperform the other approaches. For less compressed representations (i.e., larger values of f), SJL can outperform other approaches for even smaller dimension table selectivities.

In the next section, we perform a comparative analysis of the star-join query costs ($\mathcal{N}_{\text{star}}$) associated with the different index structures under study. We have shown in this section that a DataIndex-based approach and the bitmapped index/BJI approach are both among the most efficient known approaches for evaluating star-joins. In the next section, thus, we only consider these approaches and will refer to them as the SJL/DataIndex (SJL), SJS/DataIndex (SJS), and the bitmapped index/BJI (BJI) approaches. Also, to simplify the analysis, *we only consider the best performance obtainable with the BJI approach with the worst-case performance of DataIndexes*, which should be sufficient to demonstrate the superiority of DataIndexes.

4.3 Cost Comparison Preliminaries

The cost comparisons were generated based on the expressions we have presented in this dissertation for the worst- or best-case performance achievable with the different algorithms under study. The query utilized to perform the analysis is the query presented in Section 3.3 and joins the `TIME`, `CUSTOMER`, `SUPPLIER` and `SALES` tables of our sample star-schema. The query is repeated below for convenience.

```
SELECT U.Name, S.ExtPrice
FROM SALES S, TIME T, CUSTOMER C, SUPPLIER U
WHERE T.Year BETWEEN 1996 AND 1998 AND U.Nation='United States' AND
C.Nation='United States'
AND S.ShipDate = T.TimeKey AND S.CustKey = C.CustKey AND S.SuppKey =
U.SuppKey
```

The corresponding selection predicates occur on the `TIME.Year`, `CUSTOMER.Nation` and `SUPPLIER.Nation` columns, and the columns displayed in the result are `SUPPLIER.Name` and `SALES.ExtPrice`. This query is thus similar to the "Volume Shipping Query" in [138], which identifies sales volumes between different nations. Such a query is relatively typical of OLAP environments.

Finally, we continue to use the same metric as used previously, the number of blocks accessed to evaluate the query, $\mathcal{N}_{\text{star}}$. Using this metric, we first examine a baseline case where only the overall size of the warehouse is varied. We then analyze the corresponding performance sensitivities with respect to query selectivity, compression levels, and, for the SJS approach, available memory. Table 7 lists the parameter values used in the baseline analysis.

4.3.1 Baseline Case

For the baseline experiment, we assume the following parameter values:

- Selectivity on the fact table, ς_F, is 0.01 (i.e., 1% of fact table rows appear in the join result).

- Selectivity on each dimension table, ς_D, is 0.05 (i.e., 5% of the rows of each dimension table appear in the join result).

- Selectivity on each range predicate, V_{range}, is computed as $\varsigma_T|T|c$, where $\varsigma_T|T|$ represents the number of rows appearing in the join result for table T and c is the *distinctness factor* of the range selection, which we assume to be 0.2. For instance, consider the base case for the `SUPPLIER.Nation` selection predicate. If $\varsigma_D = 0.05$ and $|\text{SUPPLIER}| = 10,000$, then 500 rows from this table appear in the join result. Multiplying this number by the distinctness factor of 0.2 results in a V_{range} value of 100. Thus there are 100 distinct values in this range selection.

Holding the above values constant, we then vary the size of the database by varying the *scale factor* from 0.1 to 1000. This results in overall database sizes ranging from about 86 MB to about 860 GB. As we will soon show, the expressions derived for the

Parameter	Description	Value
$\lvert\mathcal{D}\rvert$	Number of dimensions tables involved in join	3
$\lvert\mathcal{C}_F\rvert$	Number of fact-table columns that contribute to the join result	1
B	Effective Size, in bytes, of a data block	8,000
π	Size, in bytes, of a pointer to a data block	4
r	Size, in bytes, of a RID	6
|SALES|	Number of records in SALES fact table	$6,000,000\times$ scale factor
ς_F	Selectivity factor on fact table	0.01
|TIME|	Number of records in TIME dimension table	$2,557$
|CUSTOMER|	Number of records in CUSTOMER dimension table	$150,000\times$ scale factor
|SUPPLIER|	Number of records in SUPPLIER dimension table	$10,000\times$ scale factor
ς_D	Selectivity factor on dimension table D	0.05
c	Distinctness factor of range selection	0.2
V_{range}	Number of distinct search-key values referenced by a particular range selection	$\varsigma_T\lvert T\rvert c$
w(T.Year)	Column width of TIME.Year, in bytes	4
w(C.Nation)	Column width of CUSTOMER.Nation, in bytes	25
w(U.Nation)	Column width of SUPPLIER.Nation, in bytes	25
w(U.Name)	Column width of SUPPLIER.Name, in bytes	25
w(SALES)	Table width, in bytes	131
w(TIME)	Table width, in bytes	28
w(CUSTOMER)	Table width, in bytes	269
w(SUPPLIER)	Table width, in bytes	243
f	Compression factor	0.2
M	Number of blocks allocated to input BDI	8,000

Table 7: Parameters used in the Baseline Analysis

memory requirements for BJI in (36), for SJL in (11), and for SJS in (18), indicate that BJI requires more memory than either SJL or SJS. Thus we assume in this analysis that, for a given database size, the system is equipped with sufficient memory to perform a star-join using the BJI approach. For instance, from (36), we know that a star-join using BJI for an 860 GB database requires approximately 2.5 GB of memory. We then assume that for an 860 GB database, there exists 2.5 GB of main memory. For larger databases (e.g., scale factor greater than 1000), this assumption may not be valid and so different techniques must be used, such as SJS for DataIndexes and hash-join for BJI. However, we do not consider such cases and instead focus our analysis on databases that are 860 GB or less in size. We include the performance of SJS in this range for comparison purposes. In the baseline case, we assume that 64 MB of memory is available to be allocated to the input BDI (i.e., $M = 8,000$ blocks).

The resulting plots for the baseline case are presented in Fig. 10 (Note that Fig. 10 as well as the sensitivity plots are displayed using a log scale for both axes). All three approaches exhibit a similar pattern - the cost or number of required block accesses increases as the size of the database increases. However, it is quite clear from Figure 10 that both SJL and SJS outperform the BJI approach over the entire range. In addition, the cost of the BJI approach increases much more quickly than does the cost of either SJL or SJS. This is primarily due to the fact that BDIs are maintained separately with DataIndexes and so only columns of interest need to be brought from disk. For a relatively small database, e.g., 86 MB or scale factor 0.1, SJL requires only $2,007$ block accesses, SJS requires $3,403$ accesses, and BJI requires $7,810$ such accesses. For larger databases, this difference is at least an order of magnitude. For instance, an 86 GB database (scale factor 100) requires approximately 2 million accesses for SJL and approximately 3 million for SJS, compared to 1.5 billion for BJI. The weak performance of BJI, especially with large database sizes, is largely due to the last term in (33), the expression for the cost to form the join-rowset R_{JOIN}. From

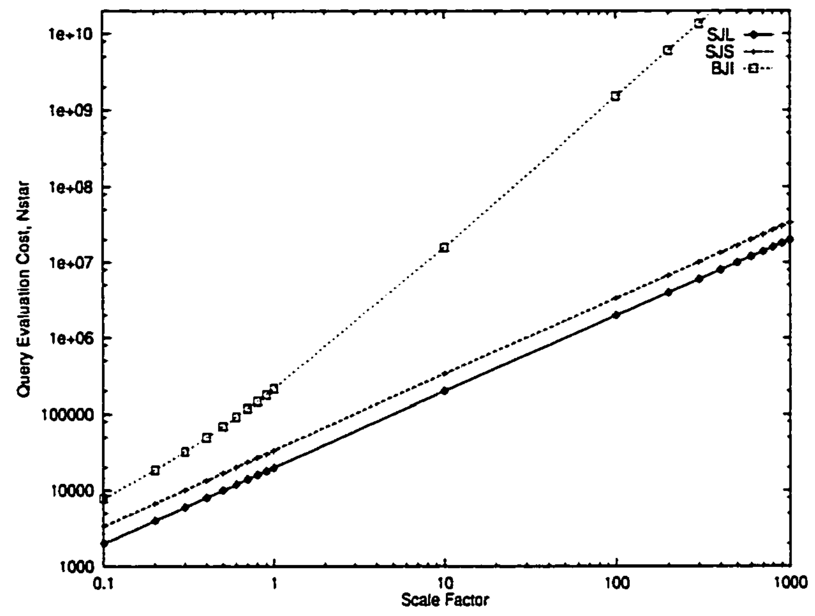


Figure 10: Base Case Performance ($\varsigma_F = 1\%$, $\varsigma_D = 5\%$ and $f = 20\%$)

this expression, we can see that BJI performs in $O(|F| \times |D|)$.

Overall, the base case curves clearly show that DataIndexes outperform the BJI approach. As we shall see, this pattern is repeated throughout the rest of our experiments.

4.3.2 Sensitivity to Query Selectivity

Query or fact table selectivity, ς_F, is an important factor in peforming a star-join for all three approaches. For SJL, ς_F impacts the cost of scanning the fact table to create the final query output, as shown in (10). For SJS, ς_F impacts the cost to restrict the JDIs (14), the cost to create the output BDIs (16), and the cost to create the final output (17). For BJI, ς_F also impacts the cost to construct the final output, as shown in (35). Based on these expressions, we would expect a decrease in ς_F (i.e., higher selectivity) to improve the performance of all approaches. In order to study this sensitivity of the different approaches to query selectivity, we repeated the baseline

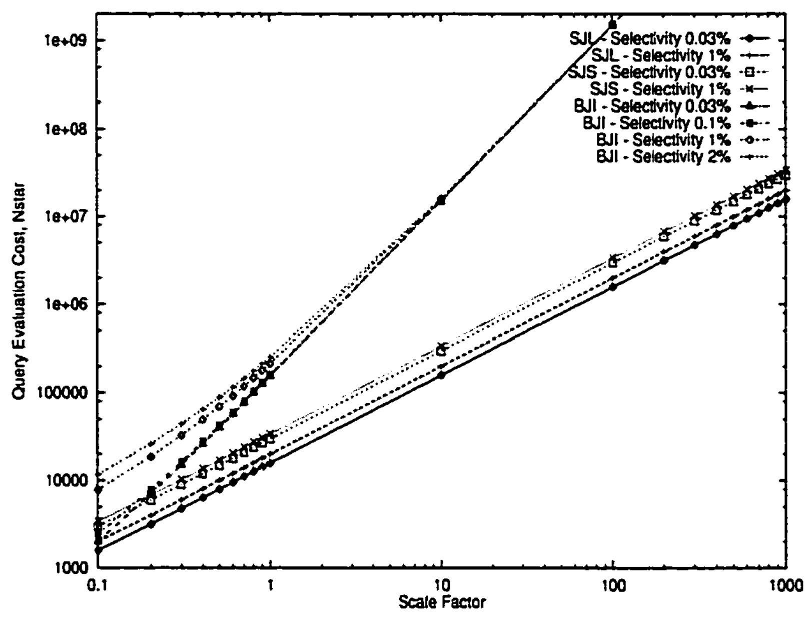


Figure 11: Sensitivity to Query Selectivity

experiments using several fact table selectivities ranging from 0.0003 to 0.1. From this analysis, it became clear that all approaches are more sensitive to relatively small values of ς_F (e.g., less than 0.0005) and that BJI is most sensitive to changes in ς_F. We therefore chose to include results for the following values of ς_F: 0.03%, 0.1%, and 2%.

The resulting plots are shown in Fig. 11.

As expected, the overall shape of the curves in Fig. 11 remains the same as in the base case, and all approaches do in fact benefit from higher selectivity. Overall, SJL still outperforms BJI over the entire range, as shown by the two lower-most curves in Fig. 11, which represent the cost of SJL for 1% selectivity (i.e., the baseline case) and 0.03% selectivity. These are the only two curves displayed for SJL because the performance of SJL is largely insensitive to changes in ς_F, except for very small values of ς_F. BJI, on the other hand, exhibits a significant improvement from lower values of ς_F, but these improvements only occur for small to medium sized databases. In fact, for very small values of ς_F and small database size, the performance of BJI approaches that of SJL. However, SJL still performs better. For instance, an 86 MB database (scale factor 0.1) and selectivity of 0.03% requires 1,587 block accesses for SJL and 1,990 accesses for BJI. However, the same selectivity with an 86 GB database (scale factor 100) requires about 1.5 million block accesses for SJL and about 1.5 billion accesses for BJI.

We now examine the curves for the SJS approach (the two curves just above the SJL curves). Like SJL, SJS is also rather insensitive to changes in ς_F. It is interesting to note, however, that for very small values of ς_F and small database size, BJI outperforms SJS. For example, an 86 MB database (scale factor 0.1) and selectivity of 0.03% requires only 1,990 accesses for BJI compared to 2,983 block accesses for SJS. As the database size increases, however, the performance of SJS surpasses that of BJI. For instance, an 86 GB database (scale factor 100) and selectivity of 0.03% requires about 2.9 million accesses for SJS compared to about 1.5 billion accesses for BJI.

From this analysis, it appears that BJI is somewhat more sensitive to changes in ς_F than either SJL or SJS. This phenomenon can be explained by examining the expressions for the cost to scan the fact table to create the final query result for SJL in (10) and for BJI in (35). Both expressions are similar in that they take the minimum of either the number of fact table rows appearing in the join result (i.e., $\varsigma_F|F|$) or the number of blocks required for all relevant fact table columns. The difference is that for BJI, the number of blocks required for fact table columns will be much greater than for SJL, since BJI requires that the entire fact table tuple be loaded for each tuple appearing in the join result. Thus the term representing the number of fact table rows will usually be the minimum term for BJI and so selectivity will typically have a greater impact.

4.3.3 Sensitivity to Compression Factor

Another important factor in performing a star-join with the approaches in this study is compression. As mentioned previously, some degree of compression is typically used in data warehouses, so it is worth examining the impact of such compression. Recall that we define the compression factor f to be a percentage representing the degree of compression, where $f = 1$ indicates no compression. Also recall that we assume no compression of BDIs, therefore, this analysis affects only the BJI approach. The level of compression appears to be a significant factor in the BJI approach, as f appears in both the $\mathcal{N}_{\text{ROWSET}}$ and $\mathcal{N}_{\text{JOIN}}$ expressions in (26) and (33), respectively. By examining these expressions, we would expect an increase in the amount of compression achieved (i.e., a decrease in f) to improve the performance of BJI. In order to study this sensitivity of BJI to varying degrees of compression, we repeated the baseline experiments with compression factors of 10% and 30%. The results are displayed in Fig. 12.

Here again the overall shape of the curves in Fig. 12 remains the same as in the base case. As expected, the performance of BJI does in fact improve when more compression is used. For instance, for a database size of 86 GB (scale factor 100) in the baseline case (20% as shown in Fig. 12), 1.5 billion block accesses are required for BJI. When more compression is applied, or f is reduced to 0.1, the number of accesses decreases to approximately 759 million. For lower compression levels (i.e., higher f), as expected, the number of block accesses increases for BJI (from 1.5 billion to 2.2 billion).

We now examine the curves for the BJI approach with respect to those of SJL and SJS. From Figure 12, it is clear that both SJL and SJS outperform BJI over the entire range. As in the baseline case, there is a significant gap between the cost curves for BJI and both DataIndex approaches, even for smaller database sizes. We

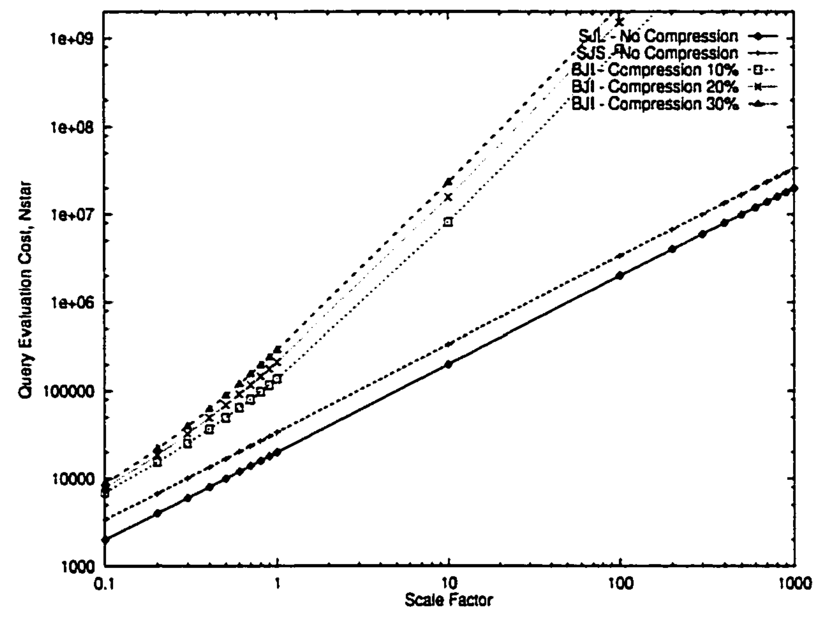


Figure 12: Sensitivity to Compression Factor

emphasize again the fact that these curves represent the best case BJI (some degree of compression) and the worst case SJL and SJS (no compression). We expect significant improvements in the performance of SJL and SJS when compression is introduced. Overall these results indicate that while increased compression levels can improve the performance of BJI somewhat, compression alone does not improve the performance enough to be comparable to the performance of the DataIndex approaches.

4.3.4 Memory Requirements

Finally, we compare the memory requirements for SJL and BJI. As shown in Figure 24, SJL requires significantly less memory than BJI on average. For instance, a database of 86 GB (i.e., scale factor 100) requires only 31 MB of memory for SJL, yet requires 243 MB for BJI. As the database size increases, the memory requirements for BJI increase much more quickly than for SJL. For example, a 344 GB database (i.e., scale factor 400) requires 125 MB of memory for SJL, yet requires 972 MB for BJI. This

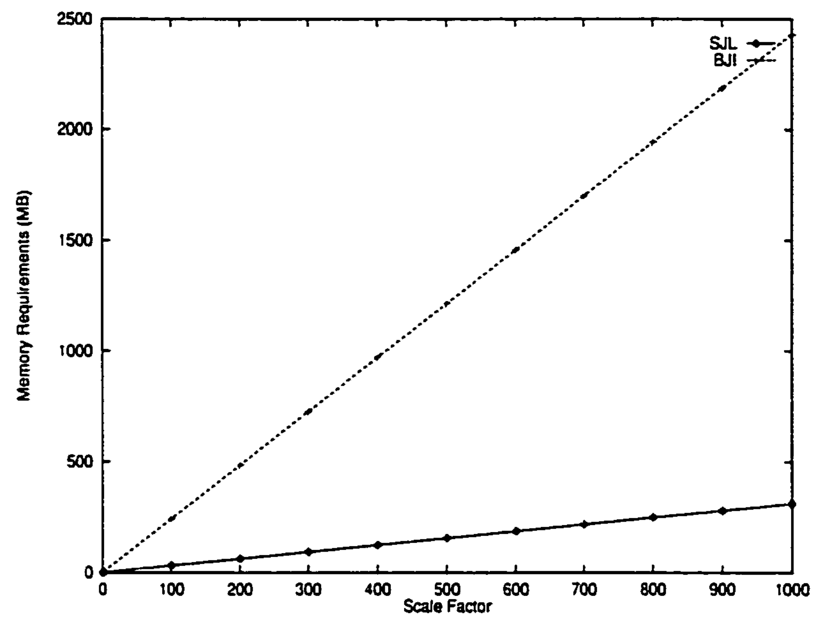


Figure 13: Memory Requirements for SJL and BJI

result is largely due to the fact that SJL loads only the columns that are relevant for the join, whereas BJI loads the entire dimension table for such columns. Even for smaller databases, SJL still requires less memory than BJI. For instance, a database of 86 MB (i.e., scale factor 0.1) requires 0.14 MB of memory for SJL and 0.28 MB for BJI. For larger databases, the SJS approach would be used, which requires only a small amount of memory, 64 KB for our example query. Recall from (18) that the memory requirements for SJS are neglible, especially when compared to the requirements of SJL and BJI. For this reason, the memory requirements for SJS are not included in Figure 24.

4.3.5 Discussion

The results of the foregoing analysis indicate that our proposed algorithms significantly outperform existing star join algorithms under conditions that are likely to occur in practice. We have also shown that our algorithms scale well when important parameters such as query selectivity are varied. Furthermore, we have also shown that our algorithms require less memory than existing techniques.

Having presented a detailed analytical comparison, we now turn our attention to an actual implementation of the proposed structures and algorithms.

Chapter 5

An Implementation of DataIndexes

In this chapter, we describe an implementation of a data warehousing system based on DataIndexes, which we refer to as *Curio*. We also present the results of a performance evaluation which compares the query processing speeds, storage requirements, and loading times of Curio with those of several existing relational data warehousing management systems (RDWMS) products.

5.1 Implementation Details

Curio is implemented in C++ and runs on several platforms, including Solaris, Linux, and Windows NT. It has both a GUI and a command-line interface and supports a large subset of the SQL standard. Thus, Curio appears to be relational on the surface. When an SQL statement is entered, it is parsed and translated into the appropriate function calls so that the Curio statement processor can execute the request. For instance, a `CREATE TABLE` statement may physically create a separate DataIndex structure for each column in the table. Metadata is maintained for each table (e.g., column names, data types), so that the notion of a table still exists when the table is queried. JDIs are created for columns that are specified as foreign keys. This metadata is stored in the Curio system catalog so that it can be used in subsequent query processing.

Data can be loaded via two methods: manually, through the GUI, or in bulk, through a bulk load utility. The bulk load utility can load data from two types of input sources: flat files or ODBC sources. The bulk load utility will build JDIs during the load process based on the schema information in the catalog.

5.2 Performance Study

We now present the results of our performance study. The products selected for the study are Oracle (version 8.0.5) [100], Red Brick Warehouse (version 5.1.5) [111], and DB2 Universal Database (version 5.0) [71]. These products were selected because their respective vendors have been identified as the market leaders in relational OLAP (ROLAP)-based data warehousing solutions. In a recent study [37], Oracle had the largest market share (31.8%), followed closely by IBM (21.7%). We also felt that Red Brick (now acquired by Informix) would be an interesting comparison as this product is a "warehouse mostly" solution (like Curio).

We first present the query processing results, since we believe query performance to be the most important aspect of this analysis. All tests were performed on a Windows NT machine having a single 300 MHz Intel Pentium processor and 64 MB of RAM (a very low-end machine). The queries used in this experiment are based on the star schema shown in Figure 14. This schema has 3 dimensions: PRODUCT, CUSTOMER, and TIME. The fact table is PURCHASE, which has a single foreign key column corresponding to each of the 3 dimensions. The numbers displayed next to each column indicate the size of the column in bytes. For each RDWMS product, indexes were built on the following columns: PURCHASE.Customer_ID, PURCHASE.Product_ID, PURCHASE.TimeKey, PURCHASE.Quantity, and CUSTOMER.Age. This scheme indexed *only those columns used in the test queries*, thus providing a significant advantage for the above mentioned products in terms of storage requirements. Typically, in a warehouse environment, indexes would be built on most (if not all) attributes, incurring significant additional storage overhead. Where possible, we used specialized indexes to give the other products as much of an advantage as possible. For example, bitmapped indexes were used in Oracle and star indexes in Red Brick.

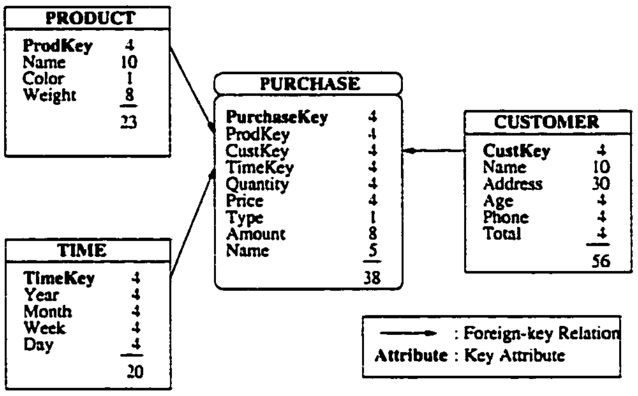


Figure 14: Warehouse Schema Used in Performance Evaluation

The data for the schema were randomly generated and loaded into each RDWMS using their respective bulk load utilities. Data loading and index creation were treated as separate steps so that loading times could be compared to Curio (since Curio does not have an index creation step). Three database sizes were considered in this experiment: 0.25 GB, 0.5 GB, and 1 GB. Database size here refers to the size of the raw data, and thus does not include any overhead that may be added once the data is loaded. Table 8 lists the number of tuples in each table for each of the sizes considered.

Table	0.25 GB	0.5 GB	1 GB
PURCHASE	5,000,000	14,000,000	22,000,000
CUSTOMER	10,000	20,000	40,000
PRODUCT	100,000	200,000	400,000
TIME	2,500	5,000	10,000

Table 8: Table Cardinalities for Performance Evaluation

The queries used in this experiment are described in Table 9. For each query, Table 9 includes a description of the query, the SQL formulation, and the *characteristics* of the query. The characteristics indicate the complexity of the query in terms of number of tables joined, number of restrictions, and number of columns projected. For instance, Query 3 joins 3 tables, has 1 fact table restriction and 1 dimension table restriction, and projects 4 columns. Note that fact table restrictions are much more costly than dimension table restrictions, since many more tuples must be evaluated.

ID	Description	SQL Formulation	Characteristics
1	List high volume purchases for a particular customer.	`SELECT Purchase_ID, Type, Quantity FROM PURCHASE WHERE Customer_ID = x AND Quantity > y`	single table query 2 restrictions 3 columns projected
2	List high volume purchases and the associated customers.	`SELECT C.Name, Quantity FROM PURCHASE P, CUSTOMER C WHERE C.Customer_ID = P.Customer_ID AND C.Customer_ID = x AND Quantity > y`	2-way join 2 restrictions: 1 fact, 1 dimension 2 columns projected
3	Find customers having high volume purchases of a particular product.	`SELECT Purchase_ID, C.Name, P2.Name, Quantity FROM PURCHASE P1, PRODUCT P2, CUSTOMER C WHERE P1.Product_ID = P2.Product_ID AND P1.Customer_ID = C.Customer_ID AND P2.Product_ID = x AND Quantity > y`	3-way join 2 restrictions: 1 fact, 1 dimension 4 columns projected
4	Find customers having high volume purchases of a particular product and the month of these purchases.	`SELECT C.Name, P2.Name, Month, Quantity FROM PURCHASE P1, PRODUCT P2, CUSTOMER C, TIME T WHERE P1.Product_ID = P2.Product_ID AND P1.Customer_ID = C.Customer_ID AND P1.TimeKey = T.TimeKey AND P2.Product_ID = x AND Quantity > y`	4-way join 2 restrictions: 1 fact, 1 dimension 4 columns projected

Table 9: Queries Used in Performance Evaluation

For each of the queries, we provide a plot of the response times for each RWDMS at each of the 3 data size levels. Response time, for the purposes of this experiment, is defined as the elapsed time (in seconds) from the start of the query until the first record is displayed. This metric was selected because the display of the first record marks the point at which an analyst may begin processing the retrieved information. The most unproductive time of a query session is the time an analyst spends waiting for the first record to appear [22], and so that is what we measure in this analysis. Note that the queries were run under controlled conditions to eliminate the effects of extraneous events. In particular, the machine was run in a "stand-alone" mode, thus rendering unnecessary the need for running any network processes. Additionally, the only processes running on the test machine other than the relevant RDWMS software processes were core OS processes. Also, to make the test fair, we eliminated the effects of caching by restarting the machine between each successive query.

The results for all tests are presented using bar charts. The numbers above each bar for Red Brick, DB2, and Oracle are ratios of the performance compared to that of Curio. We first present the results of the queries. We then follow with a comparison of storage requirements, and then a comparison of loading times.

5.2.1 Query Results

Figures 15A through 15D show the response times for each of the selected products on the 4 queries. In each of these figures, it is clear that Curio outperforms the other products, and this performance advantage increases as the raw data size increases. For Query 1 at the 0.25 GB level (Figure 15A), Red Brick appears to come close to the 16 second response time of Curio. However, Curio is still nearly twice as fast as Red Brick (29 seconds). For the large data size (1 GB), Curio (54 seconds) is approximately 10 times faster than Oracle (548 seconds) on the same query. Note that Query 1 is a single table query. For more complex queries involving joins, Curio performs especially well as shown in Figures 15B, 15C, and 15D, which are 2-way, 3-way, and 4-way joins, respectively. For instance, a 4-way join on a 1 GB database (Query 4 in Figure 15D), Curio's response time (37 seconds) is approximately 14 times faster than that of Oracle (546 seconds).

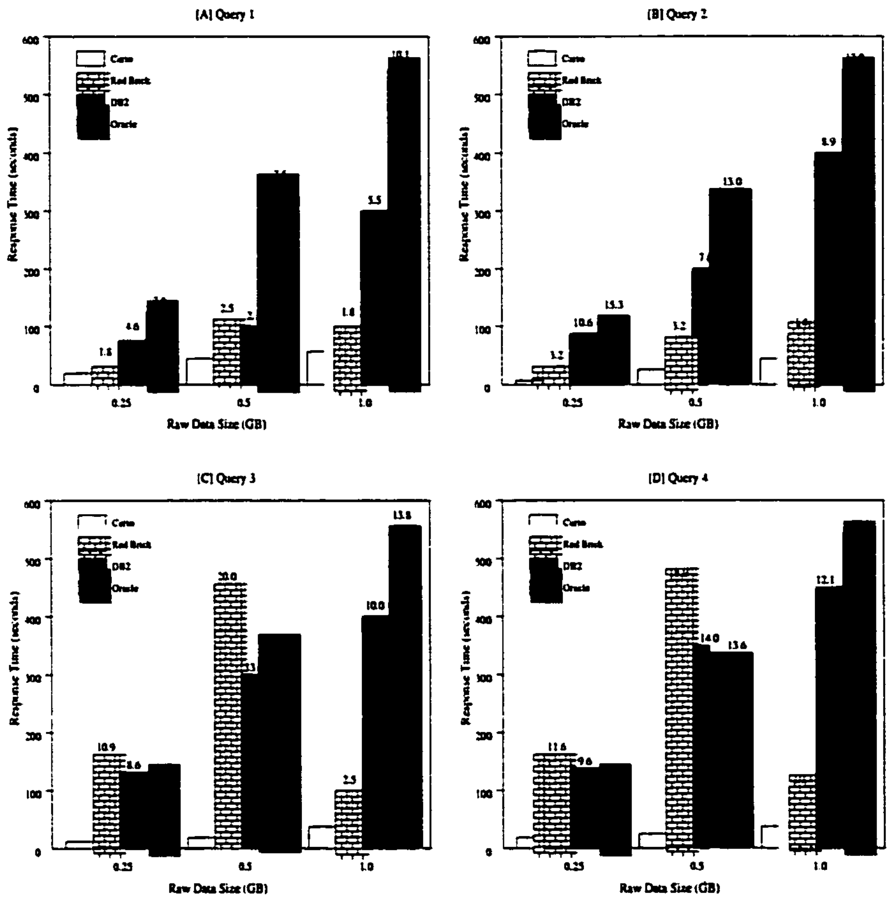


Figure 15: Response Times for Queries

5.2.2 Storage Requirements Results

We now compare the storage requirements of each product for loading the 3 sizes of raw data. The resulting database sizes for the unindexed data and the indexed data are provided in Figures 16A and 16B, respectively. From these figures, it is clear that in general, Curio is much more efficient in terms of storage requirements. For example, 1 GB of unindexed data requires nearly twice as much disk space in Oracle, as shown in Figure 16A, while 1 GB of indexed data requires 3 times the space, as shown in Figure 16B. Red Brick is clearly the second best performer in terms of storage requirements, especially for unindexed data (Figure 16A). However, once indexes are created, storage requirements increase significantly (e.g., more than 1.5 times for 1 GB of data for Red Brick), as shown in Figures 16A and 16B.

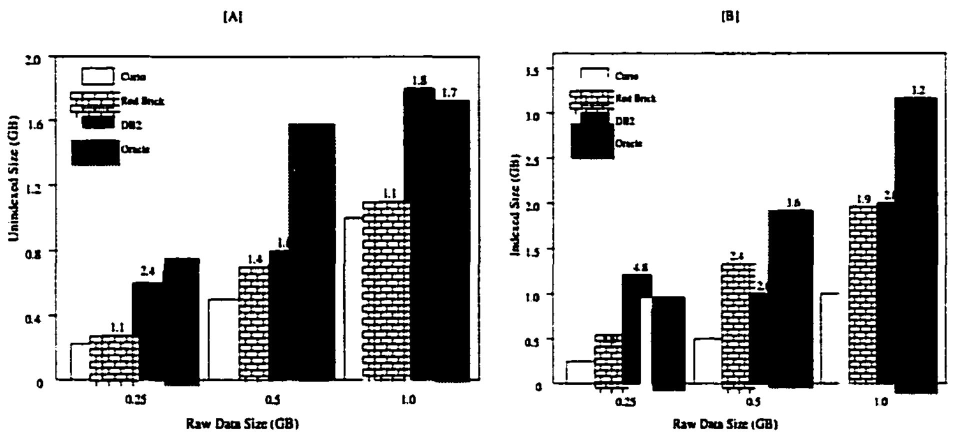


Figure 16: Comparison of Database Sizes

5.2.3 Loading Time Results

Finally, we examine the time to load the data and build the indexes for each product. Comparisons of these times are provided in Figures 17A and 17B, respectively. From these figures, it is clear that an additional advantage of the Curio design is its superior performance in loading. Without even considering the fact that Curio requires no index creation, Curio is still significantly faster than the other products. For example, Curio loaded 1 GB of data in 508 seconds, 10 times faster than Oracle (5,535 seconds, or just over 1.5 hours) as shown in Figure 17A. In Figure 17B, index creation times are displayed. Although Curio essentially has no index creation time, we have repeated the loading times for Curio on this plot. This figure shows the magnitude of the costs associated with building indexes. Of the 3 other products, the fastest creation time for 1 GB of data is Red Brick, 14,940 seconds or just over 4 hours, while the slowest time is Oracle, 45,738 seconds or over 12.5 hours.

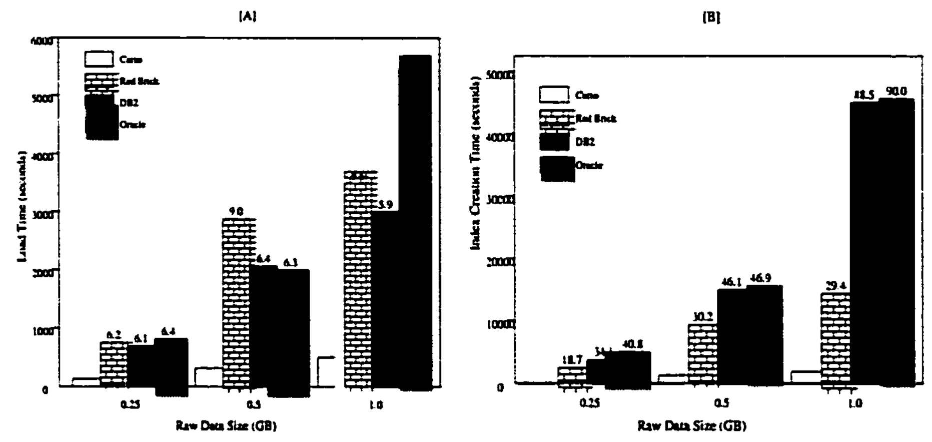


Figure 17: Comparison of Loading Times

5.2.4 Discussion

In the foregoing performance evaluation, we have compared the performance of our implementation of the DataIndex approach, Curio, to that of several leading RD-WMS products along the important dimensions of query response times, storage requirements, and loading times. For each dimension, we have shown that Curio outperforms these products, often by significant margins. These results support our analytical findings presented in Chapter 4.

Thus far in this dissertation, we have focused on the design of a data warehouse that is capable of providing interactive response times for complex queries on large volumes of data. Such a scalable data warehouse is a critical component in a warehouse of online user interaction histories. In the remainder of this dissertation, we discuss the proposed online user interaction approach and the role of the data warehouse in this approach.

Part II

The Role of Data Warehousing in Enabling Scalable Online User Interaction

Chapter 6

Online User Interaction: Overview, Survey, and Preliminaries

In this chapter, we begin our discussion of *Online User Interaction*. We begin the chapter with an overview of online user interaction. This is followed by a review of the literature in this area. Finally, we present several fundamental concepts related to online user interaction upon which our work is based.

6.1 Overview of Online User Interaction

The growth of electronic commerce during recent years has resulted in new business strategies, as well as the emergence of new technologies to support these strategies. One such business strategy that has gained momentum recently is *mass customization* [105, 106]. The idea behind mass customization is to provide each individual consumer with products and services that are tailored to his or her preferences. In his book, Mass Customization [105], Joe Pine argues that companies need to shift from the old world of mass production where "standardized products, homogeneous markets, and long product life and development cycles were the rule" to the new world where "variety and customization supplant standardized products". According to Pine, building one product is simply not enough anymore. Rather, companies need to be able to develop *multiple* products that meet the *multiple* needs of *multiple* consumers. While e-commerce has not necessarily allowed businesses to produce more products, it has allowed them to provide consumers with more choices. Instead of tens of thousands of books in a physical store, consumers may now choose among millions of books in an online store. Increasing choice, however, has also increased the amount of information that consumers must process before selecting products to purchase. To address this information overload, e-commerce sites are applying mass customization principles not to the products but to the presentation of products in their online stores [106]. In order to achieve such mass customization, many e-commerce sites are relying on *personalization* or *recommender* systems. We will refer to such systems more generally as *online user interaction* systems, since the focus of such systems is to provide online interaction with customers.

Broadly speaking, online user interaction systems are used by e-commerce sites to deliver targeted content to site visitors. Targeted content may take several forms including product recommendations, advertisements, or special offers. For instance, when a user selects a product in an e-commerce site that uses recommendation technology, the site may suggest products that are frequently purchased by customers who also purchased the selected product. *Ad targeting*, or more generally *offer targeting*, is an attempt to direct specific offers to specific consumers based upon the latter's prior behavior. Personalization systems can help e-commerce sites decide as to whom to make which offer. For example, a site can use recommendation technology to determine which banner ad to display based on keywords the consumer queried, or to which subsection of the product hierarchy a customer navigated. Recommendation technology can also help e-commerce sites implement a *one-to-one marketing* [103] strategy. One-to-one marketing attempts to overcome the impersonal nature of marketing by using technology to assist businesses in treating each consumer individually. A recommender system can help by analyzing a database of consumer preferences to overcome the limitations of segment-based mass marketing by presenting each customer with a personal set of recommendations.

Online user interaction systems can benefit e-commerce sites in a number of ways:

- **Converting browsers into buyers.** Visitors to web sites often browse a site without purchasing anything. Personalization technologies help consumers find the products they are looking for, reducing the information overload that consumers often experience. This in turn can convert browsers into buyers.

- **Increasing cross-sell.** Personalization technologies can improve cross-sell by suggesting additional products for the customer to purchase. If the recommendations are good, the average order size should increase. For instance, a site might recommend additional products in the checkout process, based on those products already in the shopping cart.

- **Building loyalty.** In e-commerce, where a site's competitors are only a click or two away, gaining consumer loyalty is an essential business strategy [113, 112]. Personalization technologies can improve loyalty by creating a value added relationship between the site and the customer. Sites invest in learning about their customers, use recommender systems to operationalize that learning, and present custom interfaces that match consumer needs. Consumers repay these sites by returning to the ones that best match their needs. The more a customer uses the recommendation system - training it on his preferences - the more loyal he is to the site.

Recent evidence suggests that online user interaction solutions can indeed have a significant business impact. For instance, Amazon.com is well known for its use of recommendation technology. In particular, Amazon's *Books* section features various types of recommendations, including **Customers Who Bought**, which recommends books frequently purchased by customers who purchased the selected book, and **Purchase Circles**, which allows customers to view the "top 10" list for a given geographic region, company, educational institution, or other organization. Amazon's extensive use of recommendation technology is believed to be the reason for Amazon's loyal customer base: 78% of Amazon's sales are from returning customers [45]. While the business impact of online user interaction solutions is itself an interesting research problem, it is beyond the scope of this dissertation and hence, we do not elaborate further on this topic. Rather, we are interested in the technological underpinnings of such systems.

The basic idea behind virtually all online user interaction schemes is to accumulate vast amounts of historical data, usually stored in a database system, and then query this historical information based on "current" visitation patterns to provide a personalized experience. Due to the sheer size of these historical knowledge bases (usually several hundred gigabytes to a few terabytes), querying the data is a very costly activity and thus takes place offline. Thus, customer profiles or recommendations are generated offline and hence are only updated whenever these offline processes run. A widely used personalization technique that follows this approach is to provide recommendations using a clustering algorithm called *collaborative filtering* [124]. Essentially, collaborative filtering places users into static groups based on their preferences (collaborative filtering will be discussed in more detail in Section 6.2.1.1).

While such pre-clustered profiling has indeed provided benefits, the static nature of this approach is problematic. A key limitation of existing recommendation technologies is that they lack the ability to adapt to changing user behavior patterns. To see this, consider an online book site that provides recommendations to its customers based on collaborative filtering. Suppose a customer, Bob, purchases a Chemistry textbook for his brother from the site. When Bob returns for subsequent visits looking for his preferred reading, Science Fiction, Bob finds his recommendation list full of Chemistry books. After a few clicks, the system should recognize that Bob is not interested in Chemistry books in his current visit and adjust its responses to be in tune with his more recent behavior. This is possible only if the system can recognize changing behavior patterns.

Another limitation of existing personalization solutions is that they are unable to scale to support the large numbers of customers and products typical of many e-commerce sites. It is not unheard of for popular e-commerce sites to have hundreds of thousands to millions of customers and even hundreds of thousands to millions of products. The approach usually taken to address this scalability problem is to provide personalization at very coarse granularities. This is typically done by placing customers into one of a small number of groups (typically on the order of 10 to 20 groups), where every customer within a particular group is presented the same content.

Referring back to the online book site example, suppose the site has 500,000 customers and 100,000 products. When the collaborative filtering algorithms run, they will produce a small number of clusters of customers who have similar purchasing preferences. Recommendations are then generated based on these clusters. Suppose cluster 1 contains 50,000 customers who have indicated a preference for Chemistry books. Each time one of these customers visits the site, his recommendation list will include selections from the Chemistry category. In fact, all of the 50,000 customers in this group will be treated the same, regardless of their unique interests or behavior. Providing recommendations at such coarse granularities clearly does not realize the potential of one-to-one interaction that is possible in an online environment.

As the above-mentioned problems with existing personalization solutions indicate, providing *true* online user interaction requires a scalable solution that is responsive to users' changing behavior patterns, and therefore reduces to performing the following tasks:

1. Tracking users' movements or behavior patterns on a site

2. Accessing a vast knowledge base that correlates specific behavior with stored knowledge, and

3. Generating responses.

A critical requirement is that responses must be generated within subsecond time frames. Clearly, this problem is one of scale. It is extremely difficult to track tens of thousands of online shoppers in near real-time, and even more difficult to access a database online and provide near real-time responses. The primary bottleneck lies in the underlying database systems that are employed to store the knowledge bases - existing database systems cannot effectively support the requirements of true online user interaction.

There are several reasons why existing database systems are unable to scale to support the requirements of online user interaction. First, the size of the underlying databases is typically vast, and may contain several types of data (e.g., navigational, transactional, demographic). Given that an e-commerce catalog may have virtually an infinite number of possible navigation paths, navigational data alone can easily result in database sizes ranging from several hundred gigabytes to a few terabytes. Second, the query operations required on this data will typically be complex, usually requiring joins of several large tables. Third, given the interactive nature of the Web, query results must be delivered within subsecond time frames. Finally, the underlying database systems must be able to provide such performance even in the presence of heavy user loads (e.g., thousands to tens of thousands of simultaneous users).

In an attempt to overcome these issues, existing online user interaction solutions rely on one of two basic approaches: (1) static profiling techniques described previously that fit customers into one of a small number of predefined static profiles (usually based on statically pre-declared information, e.g., login or zip code) and provide canned online responses, or (2) delayed, offline interaction such as email or direct mail. The former approach is the approach taken by **personalization solutions**, while the latter approach is taken by **customer relationship management** (CRM) solutions. We will describe these existing approaches in more detail in the next section. In the chapters that follow, we will present our online user interaction system which overcomes the above-mentioned limitations of existing solutions.

Having provided an overview of online user interaction, we next review the work in this area that is relevant to this dissertation.

6.2 Survey of Online User Interaction

Significant work has emerged recently within the realm of online user interaction. In this section, we review the work that is relevant to our research, which we classify into three broad categories: 1) web personalization, 2) customer relationship management (CRM), and 3) models for user interaction.

6.2.1 Web Personalization

Within the academic literature, web personalization is emerging as a major field of research. The primary objective of web personalization systems is to provide useful recommendations to customers based on analysis of purchase and preference data. Thus, we will use the terms *personalization system* and *recommender system* interchangeably. At present, there are two main approaches to personalization: *collaborative filtering* and *data mining*. Each of these approaches is now discussed in more detail.

6.2.1.1 Collaborative Filtering

Collaborative filtering (CF) [124] is a means of personalization in which products are recommended to a target customer based on the opinions of other customers. CF recommender systems employ statistical techniques to find a set of customers, referred to as *neighbors*, that have a history of agreeing with the target user (i.e., they either rate different products similarly or they tend to buy similar sets of products). Once a neighborhood of users is formed, these systems use several algorithms to produce recommendations. CF-based recommender systems generally involve three main components: *representation*, *neighborhood formation*, and *recommendation generation*.

- *Representation.* This component deals with the scheme used to model the products that have already been purchased by a customer. In typical CF recommender systems, the input data is a collection of historical purchasing transactions of n customer and m products. It is usually represented as an $m \times n$ customer-product matrix, R, such that $r_{i,j}$ is 1 if the ith customer has purchased the jth product, and 0, otherwise. Although this representation is conceptually simple, it has limitations associated with it. For instance, in practice, the customer-product matrix is typically quite sparse, which can adversely impact the ability of nearest neighbor algorithms to make recommendations. Techniques to address this problem are proposed in [120, 53]. In addition, nearest neighbor algorithms require computation that grows with both the number of customers and the number of products. Thus, given that web-based recommender systems must often deal with millions of customers and millions of products, scalabilty is a serious issue. In [119], techniques to address these issues are proposed and analyzed.

- *Neighborhood Formation.* This component focuses on the problem of how to identify the other neighboring customers. This step is the model-building or learning process and computes the similarity between customers. This is accomplished by computing an $n \times n$ similarity matrix S, where $s_{i,j}$ represents the proximity between customer i and customer j. Proximity is typically measured using either the correlation or cosine measure [119]. The neighborhood is then formed using one of several formation schemes, such as the *center-based* scheme [119].

- *Recommendation Generation.* This component focuses on the problem of finding the *top-N* recommended products from the neighborhood of customers. There are several methods available for performing this step. For instance, the *Most-frequent Item Recommendation* technique performs a frequency count of the products for all neighbors, sorts the products by frequency, and then returns the N most frequent products. Another method for generating recommendations is *Association Rule-based Recommendation*, which is based on a widely used technique in data mining, *association rule mining*. Association rule mining will be described in more detail later in this section.

Several companies offer web personalization products based on CF recommender systems, including Net Perceptions [104], Engage [136], and Macromedia (LikeMinds) [9].

While CF recommender systems have met with some success, there are also many limitations associated with them. For instance, these systems rely on subjective ratings and require that a large number of users participate in the rating effort in order to have useful recommendations. In addition, the rating schemes are best suited for homogenous product environments, such as books or CDs. For more complex products, such as computers or automobiles, rating schemes are more difficult to implement due to the large number of attributes associated with these products. Still another limitation is that the recommendations that are generated from such systems are relatively static. In other words, once a customer is associated with a neighborhood, that association remains fixed for some period of time, making it difficult to capture changing behavior. Finally, as mentioned previously, CF recommender systems suffer from serious scalability problems.

6.2.1.2 Data Mining

Data mining techniques are also used in web personalization. The mining of the input data is typically done offline, since mining involves processing of extremely large data sets and is thus quite expensive. The ouput of the mining process, which is often in the form of rules which govern user purchasing behavior, is then used to personalize content.

Much of the work in this area focuses on discovering user navigation patterns from web logs for the purpose of serving targeted or personalized content [19, 129, 147]. This area is often referred to as *web usage mining*. In [35], a detailed description of data preparation methods for mining web browsing patterns is presented. [131] proposes a web mining usage tool.

While web server logs are commonly used for web usage mining, there are limitations with these logs. In particular, browser caching and proxy servers make it difficult to identify user sessions in web logs. Most web browsers cache pages that have been requested. As a result, when a user hits the "back" button, the cached page is displayed and the repeat page access is not recorded in the web server log. Proxy servers provide an intermediate level of caching and create even more problems with identifying site usage. In a web server log, all requests from a proxy server have the same identifier, even though the requests potentially represent more than one user. To address these issues, [122] takes the novel approach of placing a user profiler on the client side using a remote Java applet. The applet tracks all requests, thus providing more accurate browsing data.

Much of the work in web usage mining is based on fundamental work in data mining for association rules [5] and sequential patterns [6, 132]. Since our online user interaction solution is also based on some of these data mining techniques, we now provide an overview of some of these concepts.

Association rule mining is commonly used in e-commerce. The main objective of this technique is to find association rules between a set of co-purchased products. Essentially, this technique is concerned with discovering association between two sets of products such that the presence of some products in a particular transaction implies that products from the other set are also present in the same transaction. More formally, let us denote a collection of m products $\{P_1, P_2, \ldots, P_m\}$ by $\mathcal{P}$. A *transaction* $T \subseteq \mathcal{P}$ is defined to be a set of items or products that are purchased together. An *association rule* between two sets of products X and Y, such that $X, Y \subseteq \mathcal{P}$ and $X \cap Y = \emptyset$, states that the presence of products in the set X in the transaction T indicates a strong likelihood that products from the set Y are also present in T. Such an association rule is often denoted $X \implies Y$, where X is the *rule antecedent* (RA) and Y is the *rule consequent.* Since association rules have largely been used in the context of supermarket purchasing analysis, these rules are often called *market basket rules.*

Association rule generation is controlled by two parameters: *support* and *confidence.* The support s of a rule measures the occurrence frequency of the pattern in the rule, while the confidence c is the measure of the strength of implication. For a rule $X \implies Y$, the support is measured by the fraction of transactions that contains both X and Y. More formally,

$$s = \frac{\text{number of transactions containing } X \cup Y}{\text{number of transactions}} \tag{38}$$

In other words, support indicates that $s\%$ of transactions contain $X \cup Y$. For a rule $X \implies Y$, the confidence c states that $c\%$ of transactions that contain X also contains Y. More formally,

$$c = \frac{\text{number of transactions containing } X \cup Y}{\text{number of transactions containing } X}, \tag{39}$$

which is nothing more than the conditional probability of seeing Y, given that we have seen X. With association rules it is common to find rules having support and confidence higher than a user-defined minimum. A rule that has a high confidence level is often very important, because it provides an accurate prediction of the outcome in question. The support of a rule is also important, since rules with very low support (i.e., very infrequent) are often uninteresting, since they do not describe sufficiently large populations, and may be artifacts.

A technique that is closely related to association rule mining is *sequential pattern mining* [6, 132]. Whereas association rule mining is concerned with detecting the *presence* of items within a particular transaction, sequential pattern mining is concerned with not only the presence, but also the *sequence* of items in a transaction. In the context of a retail application, for instance, a supermarket may order transactions by time of purchase. This ordering yields a sequence of transactions. For example, $\{milk, juice, cola\}$, $\{beer, diapers\}$, and $\{cookies, milk\}$ may be such a sequence of transactions based on three visits of the same customer to the store. In sequential pattern mining, the support for a sequence S is the percentage of the total number of sequences of which S is a subsequence [46]. Referring back to our example, $\{milk, juice, cola\}$ $\{beer, diapers\}$ and $\{beer, diapers\}$ $\{cookies, milk\}$ are considered subsequences. The problem of sequential pattern mining then is to find all subsequences from a given set of sequences that meet some specified minimum support.

We will elaborate further on this technique later when we describe the use of sequential pattern mining in our approach.

There are several personalization products on the market currently that utilize rules-based approaches. These products typically allow rules to be entered manually by users or to be generated by data mining techniques. Examples of such products include Blue Martini Personalization [86], BroadVision One-to-One [135], and WebSphere Commerce Suite [39].

6.2.2 Customer Relationship Management Solutions

Another class of solutions related to online user interaction is customer relationship management (CRM) solutions. CRM solutions focus on managing the customer life-cycle by interacting with the customer across multiple channels (e.g., web, call centers, field, resellers, retail, dealer networks). Such solutions attempt to build customer loyalty by collecting customer data and querying that data to generate responses. Several vendors offer CRM products, including Seibel Systems [127], PeopleSoft/Vantive [40], and Nortel/Clarify [94].

One key difference between CRM and personalization solutions is that with CRM, responses are generated offline. For instance, an email or direct mail may be sent to certain customers based on their past purchasing behavior. Clearly, the impact of such delayed interaction is questionable in the context of the web, where competing sites are only a click away. Since such delayed interaction is not the objective of our work, we do not elaborate further on these solutions.

Having discussed related work in the areas of web personalization and CRM, we now turn our attention to a different aspect of user interaction: data models for user interaction.

6.2.3 Data Models for User Interaction

The web has provided its users with vast amounts of information, most of which is unstructured text contained in HTML documents. To be able to efficiently access this information requires a representation or model of the data. Early work in *web data modeling* focused primarily on modeling the web as a whole for the purpose of searching the web. Notable pieces of work in this area include [87] and [81], both of which present web data models aimed at allowing declarative querying of the web as a whole for the purpose of enhancing search engines.

More recent web data modeling work has focused on modeling individual web sites, many of which are intended to improve web site design and management. Most of these web data models attempt to capture the relationship between the *content*, i.e., information displayed to the user upon request, and *structure*, i.e., the organization of the site. We briefly discuss some of this work.

The model in [48] was developed to support new methods for the declarative specification of web sites. [48] presents a logical model of the content and structure of a web site in which the site structure is modeled as a schema, and a page on the site with a page schema. At run-time, web pages are created by accessing site content, stored independently in an underlying database management system, by means of declarative specification. The model in [12] also uses the notion of a page scheme to define the structure of a site. Rather than generating pages based on a schema (as in [48]), this model was developed to support deriving the structure of a page from HTML text, for the purpose of querying hypertext data and restructuring the query results into new hypertext. In [11], the authors model the organization of a web site with an object-oriented structure called a hypertree, and a set of functions that map actual pages (i.e., URLs) to the hypertree. The model presented in [102] presents a logical model of a web site's structure, as well as a presentational model for web content, developed to support a hypermedia application development tool. [15] proposes a reverse engineering approach to generate schemas of a set of HTML or XML documents retrieved from a web site. These schemas specify the metadata, content, and structural properties of the web documents.

While all of the above web data models capture the content and structure of a web site, a key aspect that is overlooked is *user interaction with the site.* User interaction with a site is a critical component of a web data model for e-commerce sites for two primary reasons:

1. **Increased Web Site Complexity.** Online user interaction technologies have increased web site complexity. In fact, the need to provide personalized content to users is a key driver behind the increased adoption of dynamic page generation technologies, such as *Active Server Pages* (ASP) [88] or *Java Server Pages* (JSP) [91]. These technologies generate web pages on the fly, in response to each user request. Pages are generated by running business logic (e.g., personalization logic), which retrieves content from a variety of sources (e.g., database systems). In this paradigm, the standard model of a web site as a set of pages no longer applies, since the space of possible dynamically generated pages is literally infinite, given the virtually infinite content space.

2. **Increased Web Site Functionality Requirements.** Providing effective personalization requires that e-commerce sites be able to relate context-sensitive web services (e.g., services that are cognizant of who a user is or what he is doing on the site) to internal data sources (e.g., information stored in customer profiles or purchase histories) in order to serve personalized web pages. E-commerce sites also require the ability to discover user navigation patterns in order to improve site organization or discover new marketing opportunities.

These requirements are all dependent on the underlying representation of the web site, or the state of a user session at the site, or both. In short, these requirements reveal the need for a model of a web site that incorporates not only a representation that supports dynamic page generation, but also the notion of user interaction with the site.

One research work that presents a simple model of user navigation is [122]. This model is a graph-based model, referred to as a *site connectivity graph.* The nodes in the graph represent pages and the edges represent hypertext links. The model defines the notion of a *user profile* as the set of links of the user's navigation and the corresponding link times. While this model is one of the first attempts to model user navigation, a key limitation of this model is that it assumes a static web site (i.e., a site in which all pages are predefined).

We are aware of one model in the literature that incorporates user interaction with a site and supports dynamic page generation [43]. This model supports the notion of a product catalog, user navigation over this catalog, and dynamic content delivery. Since our research is based on this model, we will describe this model in more detail in the next section.

6.3 Online User Interaction Preliminaries

In this section, we provide the necessary background concepts upon which our online user interaction solution is based. Specifically, we present the underlying data model and the important concepts of *action rules* and *dynamic profiles.*

6.3.1 Site Interaction Model

We have chosen to use the site interaction model proposed in [43, 142] as the underlying data model for our online user interaction system. This model allows us to model an e-commerce site as well as the interaction of users with such a site. Table 10 contains the relevant notation.

Symbol	Description
N_i	Navigation click on a link for i
B_i	Buy item i
D	Depart from the web site
CSL	Clickstream length
RA	Rule Antecedent
RC	Set of Rule Consequents

Table 10: Table of Notation

A key aspect of any e-commerce site (or, for that matter, virtually any web site) is the *product catalog*. We view a product catalog as a labelled, directed, acyclic graph in which leaf nodes represent product instances (SKUs in retail lingo) and internal nodes represent various hierarchical groupings of products. Figure 18 shows an example of a hierarchical product catalog for a fictional online bookseller, Papyrus.com (the root node of the product catalog). The site sells several categories (i.e., internal nodes in the hierarchy) of books, e.g., Fiction and History. These categories are further divided into subcategories, e.g., Historical Fiction. Leaf nodes in the tree are products, (namely, books), e.g., M_i (a specific Mystery book), HF_j (a Historical novel), and B_k (a Biography).

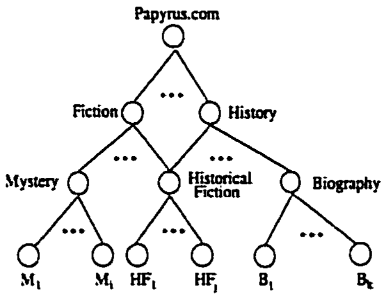


Figure 18: Product Catalog Example

A product catalog node is formally defined below.

Definition 1 Node. *A node is a generalized, abstract structure which serves as the foundation for the catalog model. A node $\mathcal{N}$ is defined as a 5-tuple $\langle L, C, P, D, A\rangle$ of descriptive information where:*

- **L** *is a unique label for the node, e.g., History.*
- **C** *is a set of* child labels, *e.g., the set of child labels for the Mystery node is* $\{M_1, \ldots, M_i\}$ *(i.e., books in the Mystery category).*
- **P** *is a set of* parent labels, *e.g., the set of parent labels for the Historical Fiction node is* $\{Fiction, History\}$.
- **D** *is a descriptor for the node, containing pointers to objects needed to represent the node in HTML, e.g., links, images and text that describe a product (a leaf node in the product catalog). These objects are* **elements**, *i.e., atomic static items that can be displayed on a web page. There are three basic types of elements (based on [12]):*
  - *An* **information element** *is a text, audio, or video object. These objects can be drawn via queries from a variety of data sources, e.g., an XML or database source.*
  - *A* **link element** *is a hypertext link. These links have two parts: (A) a text description of the link destination, and (B) a reference to one of the following:*
    1. *Another node in the product catalog (identified by its label).*
    2. *An informational page (i.e., online help, company information, customer service information, etc.), or*
    3. *An external (i.e., outside the e-commerce site) URL.*

– *A* **form element** *is a portion of a form for user entry. Form elements are collected into* forms, *which are used to facilitate user information entry (e.g., credit card information, customer service requests, etc.).*

For example, the descriptor for node M_i, a specific Mystery book, might contain a pointer to a text element containing the book's title and a pointer to an image element containing a picture of the book jacket, as well as links to the node's parent (i.e., the Mystery node), and links to other nodes representing mysteries by the same author.

– *A is a set of permissible* actions *on the node. $A \subseteq \mathcal{A}$, where $\mathcal{A}$ is entire space of possible actions on a product catalog. Six possible actions are defined on a catalog:*

1. *A* **navigation** *action is simply a click on a navigational link. For example, if user U is at the home page of an online bookseller (say, Papyrus.com in Figure 18), and wishes to see more information about Fiction, he would undertake this navigation action by clicking on the* <u>*Fiction*</u> *link.*

2. *A* **buy** *action is a click indicating the user's intention to buy an item. On an e-commerce site, this occurs when a user chooses to place an item in his shopping cart. Clearly, this action is available only from nodes which offer items for purchase. Further, an item is only available for purchase from the node that represents it, e.g., the action of purchasing Mystery M_i is only available from node M_i.*

3. *An* **un-buy** *action occurs when a user removes an item from his shopping cart, in effect undoing a previous* buy *action. This action is available at any point after the user has selected an item for purchase.*

4. *A* **check-out** *action occurs when a user completes the purchase of*

the items in his shopping cart, thereby actually creating a purchase transaction in the system. This action is available at any point at which the user has at least one item in his shopping cart.

5. *A* **form-submit** *action sends user-entered information to the web server. This type of action can be* optional *(e.g., a user can choose to respond to a survey), or* mandatory *(e.g., the user must enter credit card information and a shipping address in order to complete a* **check-out** *action, or enter a logon and password to enter a protected site).*
6. *A* **departure** *action occurs when a user departs from the site. Note that an e-commerce site's web server cannot explicitly detect a user's departure, since the HTTP request goes to another site's web server. Typically, a user is considered to have departed a web site after some threshold amount of time has elapsed since his most recent click, and thus, departure can be inferred.*

An important feature of this model is that it supports the notion of dynamically generated web pages. As discussed already, we can think of a user navigating through an e-commerce site as a user navigating over the product catalog, since the web pages are simply presentations of different views over the product catalog. Thus, a user can be said to be *located* at some node in the product catalog throughout his visit to the site. In particular, after the user's i^{th} click, he is said to be located at the node pointed to by the link chosen in his i^{th} click. Thus, if the user is located at the Fiction node before his i^{th} click, and chooses the link to Historical Fiction as his i^{th} click, then after the i^{th} click he is at the Historical Fiction node.

Based on this notion, user interaction with the site is modeled as a sequence of actions called a *clickstream*. For example, a clickstream modeling a user navigating from the root of the Papyrus.com product catalog (shown in Figure 18) to book HF_3, purchasing HF_3, and then departing from the site might consist of the following sequence: $\langle N_{Papyrus.com}, N_{Fiction}, N_{HistoricalFiction}, N_{HF_3}, B_{HF_3}, D \rangle$, where N_i denotes navigation to node i, B_j denotes buying item j, and D denotes departure from the web site.

As will become clear in subsequent sections, this model will enable us to easily represent user behavior on an ecommerce site, providing a solid foundation for our online user interaction solution. For further details regarding this site interaction model, refer to [142].

Having described the underlying data model, we now discuss the notion of *dynamic profiles* and *action rules*, which are based on this site interaction model.

6.3.2 Dynamic User Profiles

An essential component of a true online user interaction system is the ability to generate *dynamic profiles*, rather than static profiles. Generating dynamic profiles requires being able to anticipate what a user is likely to do, across several dimensions, e.g., which pages a user is likely to access, which product categories he is likely to navigate, and when he is about to leave. A dynamic profile of a user is simply a collection of information that provides a prediction of what the user's next action is likely to be. Before formally defining a dynamic profile, we first introduce the notion of *action rules*.

Action rules enable us to capture user behavior at fine granularities. In our online user interaction solution, we are interested in predicting what action a user will perform, based upon his current visitation clickstream, i.e., a sequence of actions. A clickstream representing a user's entire visit at a site is referred to as a *session*. We track users as they visit the site and log the session information. Owing to recent concerns over online privacy as well as a preference for keeping our strategies as general as possible, we consider *anonymous users* in this paper. In other words, users are identified only be a *session id* while visiting the site, and no user-specific information is maintained between user visits. Clearly, since we collect only traces of users' sessions through the site, we do not cluster users in the traditional sense. Rather, users who behave similarly can be treated similarly, resulting in a much more fine-grained interaction between a user and the site than is possible with technologies based on static profiling.

Note that session tracking is well understood technologically [19, 129, 122]. In order to record the most comprehensive tracking information, the session tracking that we will use in our solution is based on client-side tracking [122] as described previously. This approach involves sending a small Java applet to the browser to track user actions. Regardless of the technique used to create the session logs, the logs are mined to extract the *action rules*, which are of the form

$$Action_1, Action_2, ...Action_{CSL} \rightarrow Action_R;\ confidence = C\ and\ support = S$$

where $Action_i$ is a user action on the site (e.g., navigation, purchase, departure) and CSL is the (configurable) maximum clickstream length. These rules can be generated using standard sequential pattern mining tools (e.g., IBM's Intelligent Miner [38]). The result of the mining process is a set of sequential patterns of the type shown above. The antecedents of the rules have maximum length *CSL*, and correspond to certain minimum confidence and minimum support thresholds.

We now present a simple example to illustrate how session logs are mined to extract action rules. Consider the session data shown below, which is based on Figure 18.

$\langle N_{Papyrus.com}, N_{Fiction}, N_{HistoricalFiction}, N_{HF_1}, B_{HF_1}, D\rangle$
$\langle N_{Papyrus.com}, N_{Fiction}, N_{HistoricalFiction}, N_{HF_2}, B_{HF_2}, D\rangle$
$\langle N_{Papyrus.com}, N_{Fiction}, N_{HistoricalFiction}, N_{HF_1}, D\rangle$
$\langle N_{Papyrus.com}, N_{History}, N_{Biography}, N_{B_1}, D\rangle$
$\langle N_{Papyrus.com}, N_{History}, N_{Biography}, N_{B_2}, B_{B_2}, D\rangle$
$\langle N_{Papyrus.com}, N_{Fiction}, N_{HistoricalFiction}, N_{HF_1}, N_{HistoricalFiction}, N_{Fiction}, D\rangle$

Applying sequential pattern mining to this dataset and assuming a *CSL* of 3, some of the action rules that would be produced are shown in Table 11.

RA	**Consequent**
$N_{Papyrus.com}, N_{Fiction}, N_{HistoricalFiction}$	N_{HF_1}, 0.75
$N_{Papyrus.com}, N_{Fiction}, N_{HistoricalFiction}$	N_{HF_2}, 0.25
$N_{Papyrus.com}, N_{History}, N_{Biography}$	N_{B_1}, 0.5
$N_{Papyrus.com}, N_{History}, N_{Biography}$	N_{B_2}, 0.5

Table 11: Example Action Rules

As the table shows, the first rule has the highest consequent probability, 0.75, since 3 out of 4 of the sessions having sequence $\langle N_{Papyrus.com}, N_{Fiction}, N_{HistoricalFiction}\rangle$ are followed by action N_{HF_1}. This rule predicts that a user who has navigated from "Papyrus.com" to "Fiction" to "Historical Fiction" will navigate to the book HF_1 with probability 0.75 (assuming that the minimum support threshold has been met).

Having described action rules, we now present the definition of a dynamic profile (as defined in [142]).

Definition 2 *A* **Profile** *is a 2-tuple* $\langle RA, RC\rangle$ *of information where:*

1. *RA is a rule antecedent, i.e., a user clickstream of length* CSL.
2. *RC is a set* $\{c_1, c_2, ..., c_n\}$ *of rule consequents* c_i*, where each* c_i *is itself a 3-tuple* $\langle A, L, p\rangle$ *where:*

   (a) A is an action.

(b) L is a node label.

(c) p is the conditional probability of the consequent, given the antecedent.

Having described the structure of a profile, we now provide an example to make the above discussion more concrete. Consider the situation where our user Bob has arrived at the Papyrus.com home page, and has chosen the links for "Fiction" and "Historical Fiction", in that order. Based on this clickstream, i.e, $\langle N_{Papyrus.com}, N_{Fiction}, N_{HistoricalFiction}\rangle$, the system might find the following rules in the rule warehouse whose antecedent corresponds to the current visitation clickstream:

1. $N_{Papyrus.com}, N_{Fiction}, N_{HistoricalFiction} \rightarrow N_{HF_3}$ with $probability = 40$

2. $N_{Papyrus.com}, N_{Fiction}, N_{HistoricalFiction} \rightarrow N_{HF_4}$ with $probability = 20$

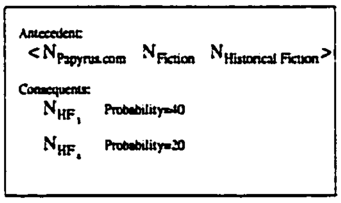


Figure 19: Dynamic Profile Example

These rules suggest that Bob will navigate to the node representing HF_3 with probability 40%, or to the node representing HF_4 with probability 20%. The contents of Bob's dynamic profile at this point are shown in Figure 19. The profile contains the rule antecedent, i.e., Bob's current clickstream, as well as the two matching rule consequents.

Our online user interaction system uses rules of this type to generate *hints* or responses about a user's next action, thus enabling a site to perform a variety of content customization. The rules are stored in a rule warehouse, which we will describe in detail in the next chapter.

Chapter 7

Design of the Rule Warehouse

In this chapter, we describe the design of the rule warehouse. We begin the chapter by discussing the semantics of the data contained in the rule warehouse. This is followed by a discussion of the logical and physical design of the warehouse. Finally, we present a comparative analysis of alternative designs for the rule warehouse.

7.1 Rule Warehouse Semantics

The rule warehouse contains a set of configurable length action rules, which are created as a result of the mining process described in the previous chapter. These rules are ultimately used to make a critical run-time decision for ecommerce sites: which content to present to a user. A rule consists of a *rule antecedent* (RA) and a *rule consequent.* An RA consists of a set of actions or behaviors $\{A_1, A_2, \ldots A_{CSL}\}$, where CSL is the clickstream length, and a rule consequent consists of a consequent action A_c, consequent node N_c, and a probability p, where p is the confidence of the rule, i.e., the conditional probability of the consequent, given the antecedent.

Each time a page request is made at a web site, the rule warehouse must serve certain information. Consider our example from Chapter 6, where our user Bob has arrived at the Papyrus.com home page, and has chosen the links for "Fiction" and "Historical Fiction", in that order (based on the product catalog in Figure 18). When Bob requests the "Historical Fiction" page, his current clickstream is $\langle N_{Papyrus.com},$

$N_{Fiction}$, $N_{HistoricalFiction}$). An ideal online user interaction system would take this representation of Bob's current behavior at the site (i.e., RA) and find all matching behaviors (i.e., rule consequents) in the rule warehouse. This information could then be used by the site to determine what content to serve Bob.

Thus, for each page request at an e-commerce site, the task of the rule warehouse can be stated as follows: *Given an RA, find all matching rule consequents.* In other words, given some behavior signature (e.g., configurable length clickstream), find all matching behaviors and their respective probabilities. Note that this query may retrieve a large set of matching consequents. Once the consequents are retrieved, the site must process them and make a content delivery decision. This processing must be done within sub-second time frames in order to provide acceptable end-to-end response times to site visitors. Thus, it is desirable for a site to be able to control the number of matching consequents that are returned for each request. Clearly, the site will want to retrieve the most useful consequents - those having higher consequent probabilities. Even though a site may control the number of matching consequents retrieved by tuning the minimum support in the data mining process, this type of tuning can only be done at the frequency with which the data mining process runs. Thus, a site should have the ability to easily configure the number of useful consequents returned for a given request. This configuration can be easily accommodated by modifying the request to the rule warehouse to incorporate a minimum threshold for the consequent probability: *Given an RA, find all matching rule consequents having a consequent probability p greater than some minimum threshold t.* The RA, combined with the set of matching rule consequents (RC) retrieved, constitutes a dynamic user profile, as defined in Chapter 6.

Our discussion thus far has focused on the representation of action rules, which represent navigational data. In addition to navigational data, personalization systems may use other types of input data such as keywords or item attributes and purchase histories [121]. Sites may use keywords or item attributes to model the customer's current interests. For instance, an online bookstore may discover an association between the *Science Fiction* and *Romance* categories, and use this association to provide recommendations. Thus, when a site visitor performs a keyword search for *Science Fiction* titles or navigates to a *Science Fiction* page, the recommendation system may suggest titles from the *Romance* category. This type of input data can easily be represented as a rule, where the RA consists of a set of attributes $\{Attr_1, Attr_2, \ldots Attr_n\}$, and the rule consequent is as defined previously. Purchase histories may also be used as input to personalization systems. Purchase histories can be represented as market basket rules, as described in the previous chapter. Such a rule would have the form $RA \implies consequent$, where the RA consists of a set of items that have been purchased by a particular customer $\{P_1, P_2, \ldots P_m\}$, and the consequent is as defined previously.

7.2 Logical Design of the Rule Warehouse

We now discuss the logical design of a warehouse to capture the semantics of action rules described in the previous section. We use the relational model to describe the logical design, since it is a widely accepted data model. We also use a ***star schema*** representation, a widely used design approach in data warehousing. Recall that a star schema typically consists of a single ***fact table*** and a ***dimension table*** for each dimension. The fact table contains foreign keys to each dimension table, along with the actual metric data. Refer to Chapter 2 for more details on the star schema design approach.

We now discuss why a a star schema is an appropriate design approach in the context of the rule warehouse. There are essentially two parts to a rule: the rule antecedent and the rule consequent. As mentioned previously, the task of the rule warehouse is to find all matching rule consequents given a rule antecedent. Thus, the rule consequent can be considered to be the metric data. Recall that a rule antecedent consists of a set of action-node pairs. Each action and node will likely have associated with it a descriptive label (e.g., the "Navigate" action label, the "Historical Fiction" node label from the product catalog in Figure 18) and possibly other information. However, storing this descriptive information with every occurrence of every action and node will be wasteful. Rather, we can use surrogate values for the actions and nodes, and store the descriptive information in smaller tables, i.e., dimension tables. Thus, in the context of the rule warehouse, a rule maps to a fact, and any associated descriptive information maps to a dimension.

Figure 20 shows one possible schema to represent rules. This star schema consists of a RULES fact table and two dimension tables, ACTIONS and NODES, which contain descriptive labels for the actions and nodes, respectively. Each record in the RULE table consists of an RA and a consequent. We assume that an RA has some maximum length n, and so an RA consists of a set of at most n action-node pairs. The consequent consists of a consequent action-node pair and a consequent probability. We assume each of these columns can be represented as a 4-byte integer. Foreign keys are defined on the consequent action and node columns so that the action and node labels can be retrieved for each rule warehouse request.

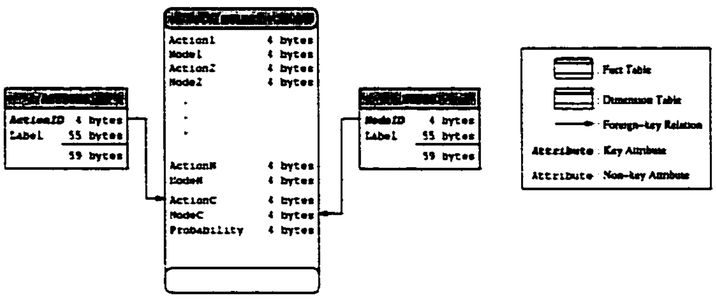


Figure 20: Action Rules Schema

Given the above schema, we now discuss the query that would be used to retrieve the matching consequents. Let $CS =< Action_1, Action_2, \ldots Action_{CSL} >$ represent the clickstream for a given user for whom we want to find all matching consequents, and let *minProbability* be the minimum threshold probability. Since this query is used to create a dynamic user profile, we will refer to this query as a *profile query*. One of the inputs to the profile query is the clickstream, which are not known until run-time. We use the following notation to represent an action i that is part of clickstream CS: _CS_$Action_i$. The *profile query* (PQ) is shown below:

```
SELECT  A.Label, N.Label
  FROM  RULES R, ACTIONS A, NODES N
 WHERE  R.ActionC = A.ActionID AND R.NodeC = N.NodeID
   AND  _CS_Action1 = R.Action1 AND _CS_Node1 = R.Node1
   AND  _CS_Action2 = R.Action2 AND _CS_Node2 = R.Node2
   AND  ...
   AND  _CS_ActionCSL = R.ActionCSL AND _CS_NodeCSL = R.NodeCSL
   AND  R.Probability > minProbability
```

The PQ is a 3-way join query on the RULES, ACTIONS, and NODES tables with a range restriction on *minProbability* and point restrictions on the ActionID and NodeID columns. One feature that is evident in this query is the potentially large number of point restrictions. The exact number of point restrictions depends on the clickstream length, CSL. In practice, CSL is expected to be fairly small (e.g., between 2 and 5). However, even in this range, this results in anywhere between 4 and 10 point restrictions (since there are $2 \times CSL$ such restrictions). When one considers the potential overhead in indexing these columns, the cost can become prohibitive (we will discuss this cost in greater detail in the next section).

To address this issue, we discuss an alternative design. If we could map every RA to a unique string of some maximum fixed length, then a single column could be used to represent an RA. Then the PQ would only require a single point restriction. This design is possible by applying the MD5 algorithm [114] to each RA to generate a unique string. MD5 is a message-digest authentication algorithm developed by RSA, Inc. The algorithm takes as input a message of arbitrary length and produces as output a 128-bit "fingerprint" or "message digest" of the input. It is conjectured that it is computationally infeasible to produce two messages having the same message digest, or to produce any message having a given prespecified target message digest. Thus, until two messages are produced having the same message digest, collisions are not an issue. The MD5 algorithm is intended for digital signature applications, where a large file must be "compressed" in a secure manner before being encrypted with a private (secret) key under a public-key cryptosystem such as RSA. MD5 is designed to be quite fast on 32-bit machines, does not require any large substitution tables, and can be coded quite compactly. Furthermore, since MD5 is a hashing algorithm, it has constant time complexity. For these reasons, MD5 meets our requirements. The alternative design based on the MD5 hash is shown in Figure 21.

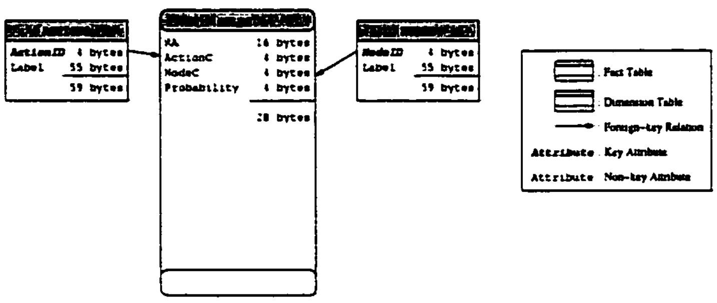


Figure 21: Alternative Action Rules Schema

If we let _CS_RA be the MD5 hash value for clickstream CS (determined at run-time), then the PQ for this design is shown below:

```
SELECT  A.Label, N.Label
  FROM  RULES R, ACTIONS A, NODES N
 WHERE  R.ActionC = A.ActionID AND R.NodeC = N.NodeID
   AND  _CS_RA = R.RA
   AND  R.Probability > minProbability
```

While the MD5 alternative can reduce the storage and query cost, it is not as flexible in terms of the queries supported. In particular, if it is necessary to support partial matches on *RA*s, then this approach becomes problematic. One possible solution is to maintain an additional mapping table containing the set of n action-node pairs keyed by *RA*. This approach would obviously introduce some overhead in terms of storage, query processing, and insertion. For our purposes, we will assume that partial matches on *RA*s are not required.

7.3 Physical Design of the Rule Warehouse

In this section, we discuss the physical design of the rule warehouse. This discussion will be based on the logical design strategies presented in the previous section. To simplify the discussion, we refer to the two logical design strategies as *Logical Design 1* (LD1) and *Logical Design 2* (LD2), where LD1 refers to the design in Figure 20 and LD2 refers to the hash-based design in Figure 21. For each logical design, we first discuss the physical design using a conventional RDBMS. We then map the relational design into an appropriate DataIndex representation (refer to Chapter 3 for a review of DataIndex physical design strategies).

In conventional relational database systems, index structures are typically defined on columns that are accessed to perform restrictions (restriction columns) and joins (foreign key columns), and on columns that are displayed (projection columns). Thus, to answer the PQ most efficiently, a conventional relational design would advocate that index structures be maintained on the restriction and projection columns and that join indexes be defined for the foreign key columns. Thus, we assume that the conventional design for LD1 will have bitmapped indexes defined on the restriction columns (the N `Action` and `Node` columns and `Probability` column in the `RULES` table) and on the projection columns (`Actions.Label` and `Nodes.Label`). We also assume that bitmapped join indexes (BJIs) are defined on the foreign key columns (`Rules.ActionC` and `Rules.NodeC`).

There are multiple DataIndex representations for any conventional design, since we can choose to store multiple columns in a single BDI. We first discuss the dimension tables, followed by the fact table. For dimension tables in general, each column is stored as a separate BDI unless it can be determined that certain columns will most often be accessed together. Thus, for LD1, we would store `Actions.ActionID`, `Actions.Label`, `Nodes.NodeID`, and `Nodes.Label` as separate BDIs. Since each of these tables has only a single result column, the key and result columns will be retrieved together. Thus, we can actually store each of these tables as a single BDI. For the fact table, each of the foreign key columns (`Rules.ActionC` and `Rules.NodeC`) is stored as a JDI. Each of the remaining columns will be stored as separate BDIs. Note that no real benefit can be gained by storing any of these columns together since these are all restriction columns, and evaluating a restriction using BDIs requires scanning the entire BDI. Thus, all blocks for the restriction columns must be loaded to evaluate restrictions, regardless of whether they are stored contiguously.

For LD2, the conventional design will have bitmapped indexes defined on the restriction columns (`Rules.RA` and `Rules.Probability`) and on the projection columns (`Actions.Label` and `Nodes.Label`). As in LD1, BJIs will be defined on the foreign key columns (`Rules.ActionC` and `Rules.NodeC`). The DataIndex representation for LD2 will be the same as LD1, except that the fact table has fewer BDIs: a single BDI for `Rules.RA` will replace the BDIs for the N pairs of `Action` and `Node` columns.

7.4 Comparative Analysis

We next perform an analytical evaluation of the design strategies proposed in the previous section. The design strategies are evaluated based on query processing cost using the expressions derived in Chapter 4. In an online user interaction environment, query processing cost is the most important criteria, since queries must provide interactive response times. Thus, as in the previous analyses in Chapter 4, the metric used is the number of blocks accessed to evaluate the query, $\mathcal{N}_{\text{star}}$. We will also discuss storage cost since it has a direct impact on query processing cost. Finally, we will also compare the approaches based on memory requirements.

The parameter settings used in this analysis are, for the most part, the same as those used in the cost comparison in Chapter 4 (refer to Table 7). The parameter settings that are unique to this analysis (e.g., column widths) are shown in Table 12.

Parameter	Description	Value
$\lvert\mathcal{D}\rvert$	Number of dimensions tables involved in join	2
$\lvert\mathcal{C}_F\rvert$	Number of fact-table columns that contribute to the join result	0
|RULES|	Number of records in RULES fact table	1,000,000× scale factor
|ACTIONS|	Number of records in ACTIONS dimension table	20
|NODES|	Number of records in NODES dimension table	1,000× scale factor
w(ActionID)	Column width of ActionID, in bytes	4
w(NodeID)	Column width of NodeID, in bytes	4
w(Label)	Column width of Label, in bytes	55
w(Probability)	Column width of Probability, in bytes	4
w(RA)	Column width of RA, in bytes	16
w(RULES)	Table width, in bytes	36
w(ACTIONS)	Table width, in bytes	59
w(NODES)	Table width, in bytes	59

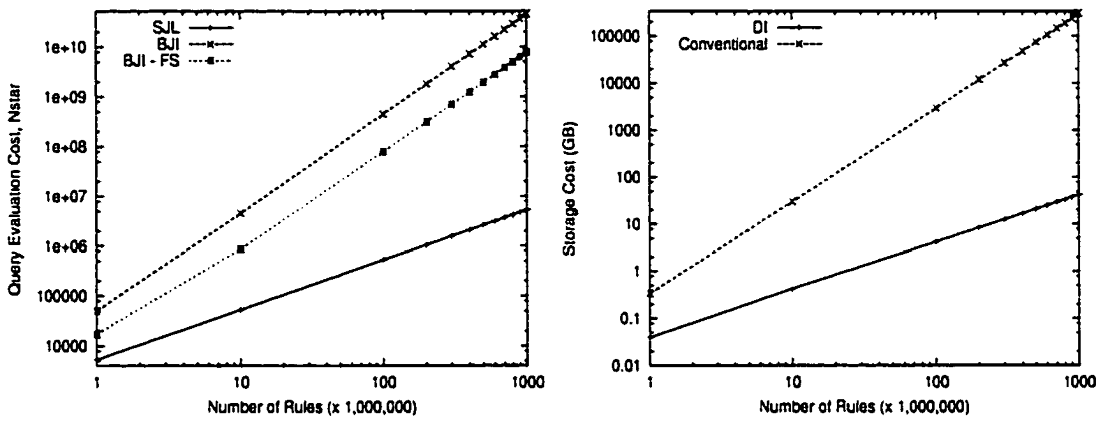
Table 12: Parameters used in the Analysis

Similar to the analyses presented in Chapter 4, we vary the size of the database while holding the other parameter values constant. More specifically, we vary the number of nodes from 1,000 to 1 million. This range was chosen to represent the size of a typical e-commerce catalog site, where the number of nodes depends largely upon the number of products in the catalog. The number of products in a large e-commerce site is typically in the range of 200,000 to 1 million. For instance, the `Amazon.de` store contains 200,000 CDs, while the `Amazon.co.uk` book store contains 1.2 million British titles [8]. The number of rules that a site may generate depends on the number of paths that can be traversed in the site, which in turn depends on the number of nodes. Based on our range for the number of nodes, we vary the number of rules from 1 million to 1 billion. Since the number of actions is quite small, the size of this table is held constant. The total size of the raw data ranges from about 0.040 GB to about 40 GB.

In this analysis, we compare the conventional design to the DataIndex design. For the conventional design, we assume that the bitmapped index/BJI approach is used to evaluate the query. For the DataIndex design, we assume that the SJL algorithm is used to evaluate the query. For the baseline case, we assume the *Logical Design 1* (LD1) strategy presented in the previous section.

7.4.1 Baseline Case

The resulting plots for LD1 are shown in Figure 22. (Note that, unless stated otherwise, all plots in this section are displayed using a log scale for both axes). We first discuss the query processing costs shown in Figure 22(a). All three curves shown exhibit a similar pattern - the cost or number of block accessess increases as the size of the database increases. As the figure shows, the SJL approach outperforms the other two approaches over the entire range. We first discuss the relationship between the uppermost curve (BJI) and the lowermost curve (SJL) curve. When the database size is small (e.g., 1 million rules), SJL provides about a 10 times improvement over BJI (about 5,000 block accesses for SJL and about 50,000 accesses for BJI). This performance improvement increases as the size of the database increases. For instance, when the number of rules is increased to 10 million, the difference is about 2 orders of magnitude. At 1 billion rules, the difference is about 4 orders of magnitude.



(a) Query Processing Cost (b) Storage Cost

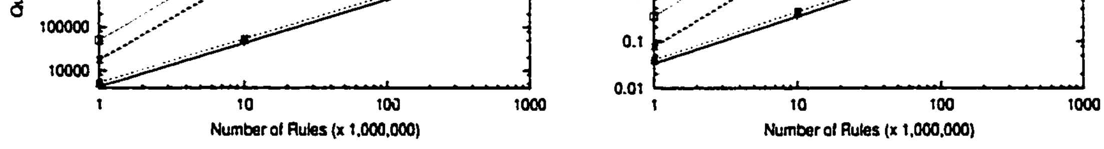
Figure 22: Cost Comparison for Baseline Case

The query cost in both approaches is dominated by the rowset creation cost, which requires loading the indexes for the 7 restriction columns. For the BJI approach, this cost is especially high due to the large size of the bitmapped indexes. As it turns out, for this approach, it is less expensive to scan the table than to load all of the indexes. The query cost for the BJI approach using a full table scan is shown as the curve labeled "BJI-FS". While this method provides improvement over the BJI approach, its performance still does not approach that of the SJL approach.

Figure 22(b) shows the storage costs (in GB) for the two approaches. The size of the DataIndexed rule warehouse (labeled "DI" in the figure) is approximately the same as the size of the raw data. The size of the warehouse based on the conventional design, however, is significantly larger, owing to the high cost of storing bitmapped indexes for the restriction and projection columns (9 columns in total).

7.4.2 Sensitivity to Warehouse Design

We now examine the sensitivity of the two approaches to a change in the design of the rule warehouse. More specifically, we examine the impact of using the *Logical Design 2* (LD2) strategy presented previously. The plots for LD2 are shown in Figure 23. Note that the cost to run MD5 is not included here due to the fact that it does not contribute to I/O cost. We first discuss the query processing costs for the LD2 design, which are shown in Figure 23(a). The overall shape of the curves for LD2 remains the same as for LD1, and again SJL outperforms BJI over the entire range. For LD2, the performance difference is not quite as large, but it is still substantial. For instance, when the database size is small (e.g., 1 million rules), SJL provides about a 4 times improvement over BJI (about 4,000 block accesses for SJL and about 18,000 accesses for BJI). When the number of rules is increased to 10 million, the difference is about an order of magnitude, while at 1 billion rules, the difference is about 3 orders of magnitude.



(a) Query Processing Cost (b) Storage Cost

Figure 23: Sensitivity to Warehouse Design

We now discuss the impact of the design change. The BJI approach is clearly much more sensitive to this design change than the SJL approach. On average, the effect of changing the design from LD1 to LD2 was to reduce the number of block accesses by about 70% for BJI. For SJL, the number of block accesses was reduced by about 20% on average. As mentioned previously, this phenomenon is a result of the large number of blocks that must be accessed in the restriction phase. By reducing the number of restriction columns in LD2, the number of data blocks that must be loaded is reduced. The BJI approach accesses significantly more data blocks than SJL, and thus realizes a greater reduction in block accesses as a result of the design change.

Figure 23(b) shows the storage costs (in GB) for the two approaches. As this figure shows, storage costs for the conventional design are much more sensitive to the design change than for the DI design. In fact, the design change results in about an 80% reduction in storage requirements for the conventional design, and about a 20% change for DI, on average. This result is easily explained. For both DI and the conventional approaches, the change results in a reduction of 8 bytes for every record in the RULES table. For the conventional design, the change also results in a reduction in the size of the index structures. Specifically, where there were originally bitmapped indexes defined on the 6 point restriction columns in LD1, there is now a single bitmapped index on the RA column in LD2. For the DI approach, there is no other impact on storage requirements since there are no additional index structures.

7.4.3 Memory Requirements

We now compare the memory requirements for the design alternatives. As shown in Figure 24, there is not much difference in the memory requirements for the two approaches, although SJL is slightly more efficient. This is due to the fact that the

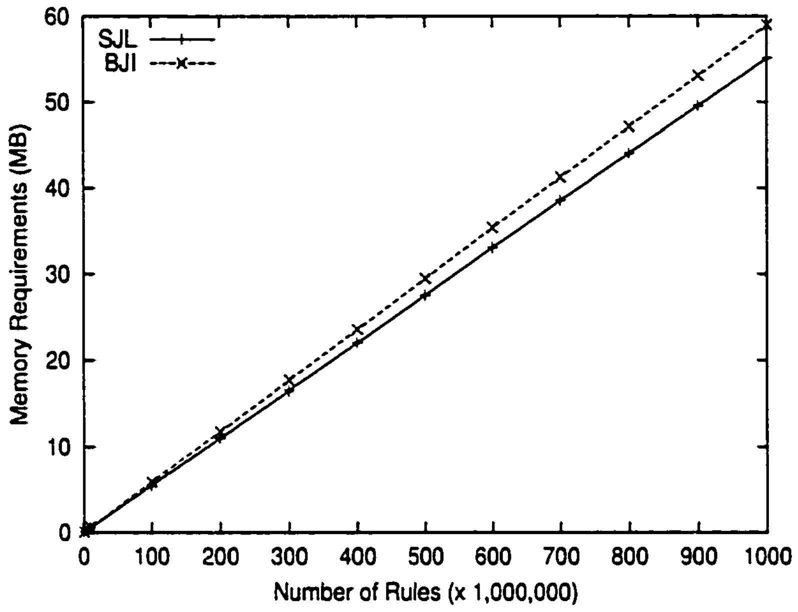


Figure 24: Comparison of Memory Requirements

width of the dimension table display BDIs (ACTIONS.Label and NODES.Label) are wide with respect to the table size. In fact, recall that the BDIs are the same width as their respective tables since we are storing these tables each as single BDIs. Still, for both approaches, the memory requirements are not substantial. For instance, the maximum database size (1 billion rules) requires only 55 MB of memory for SJL and 59 MB for BJI. Note that the memory requirements are the same for LD1 and LD2. This is due to the fact that the design change only impacts the fact table, and as **Result 1** in Chapter 4 indicates, the size of the fact table does not impact memory requirements.

7.4.4 Discussion

As the foregoing analysis shows, the DataIndex approach represents an extremely efficient alternative for the storage and retrieval of rules, and thus satisfies a key requirement for an online user interaction solution. The DataIndex approach strictly outperforms the conventional approach in terms of query processing costs, storage costs, and memory requirements, and in most cases by significant margins. We have also compared two DataIndex design alternatives, LD1 and LD2, to represent rules. Our analysis shows that LD2 is preferred over the LD1 strategy from a performance standpoint, since LD2 has lower query processing and storage costs than LD1. However, if it is necessary to be able to perform partial matches on RAs, then the LD1 approach may be preferable.

In this chapter, we have provided a detailed discussion of the design of the rule warehouse. In the next chapter, we describe the architecture of an online user interaction solution and discuss the role of the rule warehouse in this solution.

Chapter 8

Online User Interaction System Architecture

In this chapter, we present the technical details of an online user interaction system. We will refer to this system as the *Online User Interaction Server* (OUIS). We have incorporated our DataIndex technology (described in Chapters 3 through 5) into the rule warehouse component of this solution. Thus, we will focus on the role of the rule warehouse module in the overall architecture. We begin by providing an overview of the system architecture, followed by a detailed description of the system components.

8.1 System Architecture Overview

The OUIS is a component-based system which works with readily available web server and e-commerce application server systems. Figure 25 shows a graphical depiction of an "end-to-end" e-commerce system architecture, including the OUIS component.

The functionality of the system is best described using an example. Consider a user *Bob*, who clicks in an e-commerce site that utilizes OUIS technology. This causes an HTTP request to be sent to the *Web/application server* (WAS). (Note that, although the application and web servers are typically separate components, we describe them as a single component here for convenience). When the WAS receives an HTTP

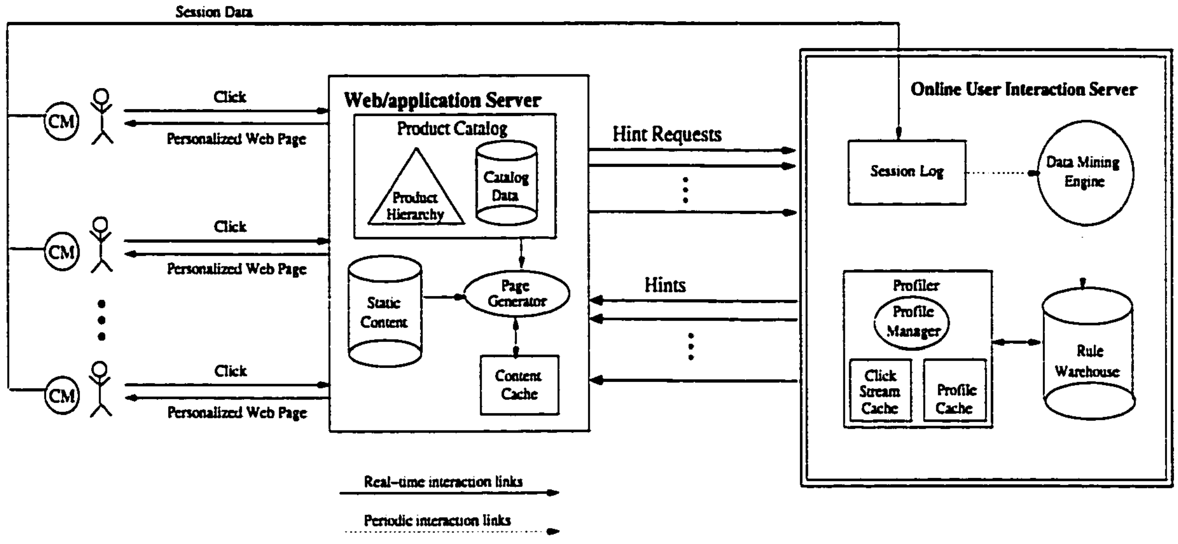


Figure 25: Online User Interaction System Architecture

request from Bob, it forwards Bob's click information to the OUIS. Upon receiving this information, the OUIS performs two tasks. First, *the OUIS updates Bob's clickstream.* Second, *the OUIS generates a hint.* A *hint* is simply a set of *action-probability pairs*, which represent actions Bob is likely to take, along with the corresponding probability that Bob will choose the action, given his current clickstream. These are drawn from the action rules described previously, where a hint consists of a set of rule consequents matching a given RA (i.e., clickstream).

When the WAS receives a hint from the OUIS, it uses the hint to generate a customized web page for Bob. Precisely how the WAS uses the hint to generate a personalized web page for a user is dependent on the needs of the web site, and is outside the scope of this dissertation. Typically, the WAS runs a set of *scripts* that describe how a web page should be generated in different situations. In an OUIS-enabled system, these scripts include business logic to handle OUIS hints.

We now provide an example to illustrate the use of hints. Consider the case where our user Bob has navigated through the sequence of product catalog nodes,

$\langle$*Fiction*, *Thriller*, *LegalThriller*$\rangle$. Further assume that upon Bob's click on the *LegalThriller* node, the OUIS returns a hint to the WAS, suggesting that Bob is highly likely to be interested in fusion jazz, based on his recent behavior. As a result, the business logic in the WAS scripts may send Bob a web page offering a discount on selections from the *Fusion Jazz* category of its online music store. Continuing with this example, suppose that Bob has navigated through several more product catalog nodes, but has still not purchased anything. In this case, on Bob's next click, the OUIS may return a hint to the WAS, suggesting that there is a high likelihood that Bob will depart in his next click, given his current clickstream. At this point, the business logic in the WAS scripts might send Bob a web page that includes a special offer, perhaps free shipping or a percentage discount on his order, to entice him to stay at the site and continue shopping.

In addition to the above interaction, each user click causes a client-side *Clickstream Monitor* to send user session (i.e., click) information to the OUIS, where it is stored in the Session Log for later use in updating the rulebase. (Note that this interaction is entirely separate from the user's interaction with the WAS). The CM module is essentially a client-side Java applet that sends clicks (including client-side cache hits) to a server, as described in, say [122]. This method of tracking session information allows the OUIS to obtain a complete account of a user's actions at the site, including those requests served by a client-side cache. Since the information stored in the Session Log is used for refreshing the rulebase (an offline operation), there is no requirement for this interaction to occur in real time.

Having described the interaction between the main components (i.e., the WAS and the OUIS) of the system, we now provide a detailed description of these components.

8.1.1 Web/Application Server

The OUIS works in conjunction with existing web servers (e.g., Apache, Netscape) and e-commerce application servers (e.g., BEA's WebLogic or IBM's WebSphere) to provide the functionalities we have described. When the WAS receives an HTTP request from a user, it forwards the user's clickstream information to the OUIS, signalling the OUIS to generate a hint. The OUIS responds with a hint, which is sent to the *Page Generator* module of the WAS. The Page Generator generates a personalized page for the user, by integrating content from various sources and formatting the content as necessary (e.g., into HTML pages). Page content is drawn from various sources, including the *Product Catalog* (which stores product information in hierarchical format as well as detailed information about each product), a store of *static content* (i.e., content not associated with a particular product catalog node, e.g., a corporate logo), and a *server cache*. Communication between the OUIS and WAS follows a client-server paradigm, i.e., the WAS requests hints from the OUIS, and the OUIS provides hints in response.

8.1.2 The Online User Interaction Server Components

The OUIS consists of two main components: (1) the *Profiler* and (2) the *Rule Warehouse*. We next provide an overview of each of these components and their interaction.

The *Profiler* is essentially a caching module and consists of three components: (1) a *Clickstream Cache*, which stores current clickstream information for each user in the system, (2) a *Profile Cache*, which stores recently-used profiles, and (3) a *Profile Manager*, which generates hints for the WAS.

We return to our example to illustrate the mechanics of the hint-generation process. Consider a situation where the WAS has submitted a *hint request* to the OUIS for Bob's i^{th} click. The Profile Manager first checks the Clickstream Cache to find

Bob's previous clickstream (if any), and updates that to include Bob's latest reported action. After determining Bob's current clickstream, the Profile Manager checks the Profile Cache for rules matching Bob's clickstream (i.e., a profile whose RA matches the observed clickstream). If such a profile is found, the Profile Manager generates a hint from it, and sends the hint to the WAS. If, at this point, another user, say *Alice*, were to follow on the same path as Bob, the reader can easily see that the needed profile would be in the Profile Cache, assuming Alice follows Bob closely enough that the profile has not been replaced by a newer profile.

If a matching profile is not found in the Profile Cache, the Profile Manager requests the information from the *Rule Warehouse*. The Rule Warehouse is a DataIndexed-based data warehouse that stores action rules, as described in Chapter 7. After the Rule Warehouse returns the matching rules, the Profile Manager generates a hint, and sends it to the WAS. This profile will also now reside in the Profile Cache until a decision is made to discard it.

The rules stored in Rule Warehouse are generated by a data mining engine using sequential pattern mining. The data mining engine takes a *Session Log*, i.e., click-stream and transaction information generated by various users clicking in the web site, as input, and generates a set of rules as output. Click and transaction data is added to the session log in real time (as noted by the solid lines in Figure 25), while the mining of the session log and update of the Rule Warehouse takes place offline (shown by the dotted lines in Figure 25). The Rule Warehouse is periodically updated, via an offline process, to incorporate recent behavior.

8.2 Technical Details

In this section, we describe the underlying technical details of the OUIS. We first present the details of the Profiler component. This is followed by a discussion of the role of the Rule Warehouse and its interaction with the OUIS components.

8.2.1 Profiler

Figure 26 provides a graphical overview of the technology that forms the basis of the Profiler module of the OUIS. We first describe the interaction between the different components of the Profiler. This is followed by a detailed discussion of each component.

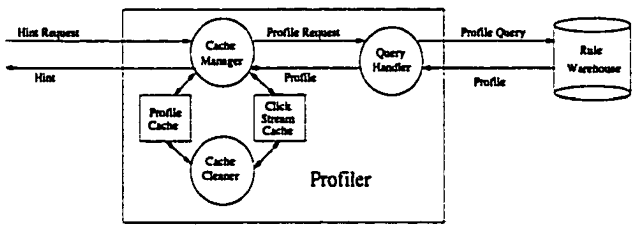


Figure 26: OUIS Profiler Module

The Profile Manager shown in Figure 25 in Section 8.1 consists of three modules, the *Cache Manager*, the *Query Handler* and the *Cache Cleaner*. We describe each of these modules in turn.

The *Cache Manager* monitors incoming hint requests, generates hints from profiles, and maintains the *Profile Cache* and and *Clickstream Cache* structures. The details of the cache structures are described later in this section.

For *cache misses*, i.e., situations where a needed profile is not found in cache, the Cache Manager generates a *Profile Query* (PQ) to retrieve the needed profile from the Rule Warehouse. When the PQ has been processed, the Cache Manager receives a profile in response. This profile is then used to generate a hint and update the Profile Cache, as described in Section 8.1.2.

The *Query Handler* handles the communication between the Profiler and the Rule Warehouse. The Query Handler accepts PQs, formulates the appropriate queries (in the format described in Chapter 7), and submits the queries to the Rule Warehouse. The Query Handler formats each query result as a profile, and forwards it to the Cache Manager.

The *Cache Cleaner* removes outdated information from both the Profile Cache and the Clickstream Cache. The idea behind the Cache Cleaner is similar to the clock hand idea in classical operating systems [134].

8.2.1.1 Profile Cache

We begin our discussion of the Profile Cache with a note on the basic structure of the cache. The cache consists of a (configurable) number of variable-sized elements, called *Cachelines* (CL). Each CL can store at most one instance of a profile. The Cache Manager accesses a CL through a hash index on the *RA* (i.e., antecedent) portion of each CL's profile. For convenience, we remind the reader that a profile consists of an *RA* and an *RC*, where an *RC* is the set of rule consequents matching the *RA* (refer to Chapter 6 for the precise structure of a profile).

CLs have three possible states:

In the **empty** state, a CL contains no profile. All CLs are *empty* at system startup. A CL returns to this state when is has been selected for replacement, before a new profile has been stored in it.

In the **requested** state, the CL is awaiting a response to a PQ. Here, the *RA* portion of the profile has been instantiated, but the *RC* (i.e., consequent) portion has not. A CL moves from the *empty* state to the *requested* state when the Cache Manager receives a profile request, but does not find the needed information in the Profile Cache, i.e., when a query must be submitted to the Query Handler. The CL remains in the *requested* state until the Query Handler receives the needed response from the Rule Warehouse and instantiates the RC portion of the profile.

In the **full** state, the CL contains both the RA and RC portions of the profile. A CL enters the *full* state when the Cache Manager receives a profile from the Query Handler (in response to a query), and remains in this state until it is selected for replacement.

A key aspect of any caching system is cache management. In particular, given a constrained cache size, cache replacement is especially critical. While there are many well-known cache replacement policies that can be used to control replacement for the Profile Cache, the OUIS employs a predictive replacement policy which exploits user navigation patterns [142]. This replacement policy is enabled by performing *lookahead queries* on the Rule Warehouse. We later discuss these lookahead queries in more detail. For now, we now describe the replacement policy.

As discussed previously, the Cache Manager stores recently requested profiles in the Profile Cache. The utility of caching such information is clear. Consider, for example, the case of two users, Bob and Alice, who are both interested in the same item, book B. In this scenario, Bob enters the site at the home page, and navigates through the product hierarchy to the page offering book B. Alice enters the site just after Bob arrives, and navigates the same click sequence. Clearly, since the profiles used to generate hints for Bob are likely to be located in the cache at this time, these profiles can be used for Alice's as well, saving the expense of querying the warehouse. This is the intuitive basis of the cache replacement policy employed by the OUIS.

In order to ensure that useful profiles are retained in the cache, the Profiler needs to be able to detect situations in which one user is following behind another, and utilize this knowledge in the cache replacement policy. For this purpose, the notion of *following distance* between two CLs is defined.

Definition 3 *Consider two CLs C_i and C_j, containing profiles P_i and P_j, respectively. Assume that these CLs and profiles exist in the context of an e-commerce site with a catalog T. The* **following distance** *from C_i to C_j, denoted by $F(C_i \rightarrow C_j)$, is the minimum number of actions required by a user whose clickstream sequence currently matches P_i to match P_j, in the context of T.*

For example, consider a situation where a user has traversed the click sequence $\langle N_a, N_b, N_c, N_d \rangle$. The user's traversal of this sequence has caused profiles P_i and P_j to be loaded into the profile cache (PC), where $P_i.RA = \langle N_a, N_b, N_c \rangle$ and $P_j.RA = \langle N_b, N_c, N_d \rangle$ (assuming a clickstream length of 3). Assume that P_i and P_j reside in CLs C_i and C_j, respectively. Clearly, $F(C_i \rightarrow C_j) = 1$, because a user with profile P_i, need only click on link c to be ascribed to profile P_j.

This notion must be extended to include the possibility that multiple users might be traversing the same click sequence at the same time. To accommodate this situation requires identifying the user that is *closest* to a particular CL in a particular click sequence. This concept if referred to as the *lowest following distance* (LFD) of a CL.

Definition 4 *Consider a set of CLs $\{C_1, C_2, \ldots, C_n\}$. The lowest following distance of a CL, say C_i, is denoted by* LFD*(C_i) and is given by:*

$$\text{LFD}(C_i) = min[F(C_1 \rightarrow C_i), F(C_2 \rightarrow C_i), \ldots, F(C_n \rightarrow C_i)]$$

LFD is computed online for a CL by *broadcasting* a user's position to each CL that is within a configurable *maximum lookahead distance* (MLD) ahead of a user's current CL. Specifically, with each user click, the Profiler updates the distance for each CL within MLD ahead of the current CL (based on permissible actions on each node), where MLD is a configurable parameter. For a CL with an undefined LFD, i.e., where no user will need it within MLD clicks, the CL's LFD value is set to MLD.

Broadcasting LFDs occurs via a set of CL pointers associated with each node. Specifically, CL C_i contains a set of pointers to other CLs, corresponding to the links available from the last node in C_i's antecedent. For example, consider the situation where C_i holds profile P_i, with $P_i.RA = \langle N_a, N_b, N_c \rangle$ and CL C_j holds profile P_j with $P_j.RA = \langle N_b, N_c, N_d \rangle$. Let us further assume that a user can traverse the click sequence $\langle N_a, N_b, N_c, N_d \rangle$. In this situation, there exists a pointer from C_i to C_j, denoting that C_j is *reachable* from C_i, shown graphically in Figure 27.

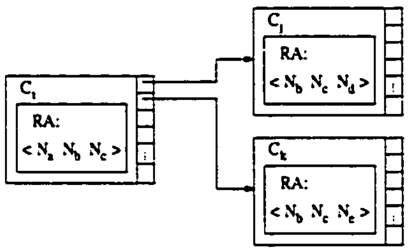


Figure 27: CL Pointer Example

The Cache Manager updates these CL pointers for each newly instantiated CL, i.e., when a CL enters the *full* state. For example, consider the situation where C_j holds profile P_k with $P_k.RA = \langle N_b, N_c, N_e \rangle$, and where a profile P_i, with $P_i.RA = \langle N_a, N_b, N_c \rangle$ is about to be added to the cache into CL C_i. We assume, for the purposes of this example, that a user can traverse the click sequence $\langle N_a, N_b, N_c, N_e \rangle$. In this situation, the Cache Manager sets a pointer in CL i pointing to CL k, as shown in Figure 27.

Having laid the groundwork for the cache replacement policy, we move on to discuss this policy in detail. We note first that the most useful CLs in the cache are those with low LFDs, since these are the items likely to be needed soonest. Thus, the cache replacement policy should take following distance into account in selecting CLs for replacement. However, in addition to LFD, it should also take time into account. In particular, CLs with low LFDs should be given priority over CLs with higher LFDs, and for CLs of the same LFD, CLs that have been used most recently should be given priority over "older" CLs, i.e., those that have not been used as recently.

This cache replacement policy is implemented using a series of *priority queues*, shown in Figure 28 as *Cache Queues* (CQ), numbered 1 to MLD. Each queue represents a specific LFD, e.g., CQ_1 stores CLs with a LFD value of 1, CQ_2 stores CLs with a LFD of 2, and so on. Each queue is ordered by time, with the oldest CL, i.e., the least recently used CL, at the front of the queue.

These cache queues are maintained in the following manner. As a user navigates on a click sequence, the LFD values for CLs on that sequence will decrease. When a CL's LFD values changes, it is moved to the end of the appropriate cache queue for its new LFD value. When a CL is referenced (i.e., a *cache hit* occurs), the CL is moved to the end of the queue, thus maintaining the queue's time ordering.

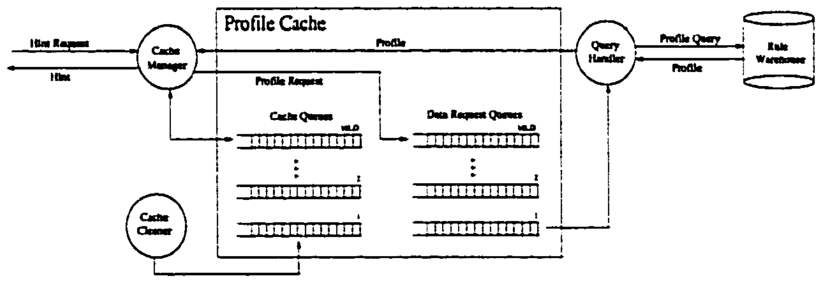


Figure 28: Profiler Cache Detail

CLs are replaced first based on LFD. More specifically, CLs are first chosen for replacement from the MLD queue. If there are no CLs in the MLD queue, CLs are replaced from the (MLD - 1) queue, and so on. Within each queue, priority is given to CLs in most-recently-used order, choosing the least recently used CL for replacement first. Since each queue is ordered based on time, the CL at the front of the queue is always the least recently used, and is the first chosen for replacement.

Note that this prioritization scheme leaves open the possibility that CLs may sit unused for a long time in a cache queue. This occurs when a CL's distance is updated because it is on a possible future path for a user. If the user chooses an alternate path, the above prioritization scheme has no means of resetting the CL's distance value to the default value. This situation is handled by noting the most recent access time of a CL with a timestamp. The Cache Cleaner module periodically checks all CLs in the cache, and removes CLs with timestamps more than T_{max} time in the past, where T_{max} is a configurable threshold value.

8.2.1.2 Clickstream Cache

As noted above, the Clickstream Cache stores user clickstreams for users active on the site. A Clickstream Cache element has a very simple structure – it need only store a userID and the most recent CSL user clicks. Since Clickstream Cache elements are simple data structures and are not reusable for multiple simultaneous users, an off-the-shelf technology, such as an object database, can be used as the caching mechanism.

8.2.2 Rule Warehouse

As mentioned previously, the Rule Warehouse is a DataIndexed-based data warehouse that stores action rules. The rule warehouse is designed using one of the design strategies presented in Chapter 7. Since we have described the technical details of the warehouse in previous chapters, we do not elaborate further on this aspect. Rather, we focus on the role of the Rule Warehouse in the overall system architecture and its interaction with the other system components.

8.2.2.1 Interaction with the Profiler

The Rule Warehouse is used to serve two types of requests from the Profiler: (1) requests for *cache misses*, i.e., requests made when a needed profile is not found in the Profile Cache, and (2) *anticipatory requests*, i.e., requests for profiles that users are likely to need. The latter type of request refers to the lookahead queries used to support the predictive cache replacement policy described in Section 8.2.1.1. These profile requests are simply CLs in the *requested* state, queued for processing by the Rule Warehouse. Both types of requests use the same profile query (as presented in Chapter 7). The main difference is the nature of the two types of queries - requests for cache misses are needed with certainty whereas anticipatory requests are needed with some probability (and perhaps may never be needed).

For this reason, the profile requests must be prioritized. In particular, requests for cache misses should be processed before anticipatory requests. In addition, priority should be given to requests for profiles that are likely to be needed in a user's next click over those likely to be needed after 2 clicks, and so on. Like CLs containing fully instantiated profiles, request CLs are assigned LFD values. Here, the LFD value refers to how soon the profile is needed. Thus, a request generated by a cache miss has an LFD value of 0, indicating that it is needed immediately, while an anticipatory request for a profile that a user can reach in 1 click has an LFD value of 1, and so on.

Request CLs are prioritized in *Data Request Queues*, shown graphically in Figure 28. As each request CL is generated, it is placed at the end of the data request queue matching its LFD value. The Query Handler module services requests in increasing priority order; within each queue, requests are serviced in first-in-first-out order. Note that this queuing mechanism may generate requests for information that is never needed, e.g., if a user chooses a different path from the path the Profiler predicted. Here, a cleaning mechanism similar to the Cache Cleaner (described above) is used to remove unneeded requests.

For both types of profile requests, the Rule Warehouse must provide extremely fast responses. A profile is used to determine the content that is served for a given request, so page generation cannot be completed until a profile has been served. Thus, any delay in this process will delay the delivery of the user's web page. In the next chapter, we will examine the overall system performance in terms of response time. In particular, we will show how a DataIndexed warehouse can significantly outperform a conventional relational DBMS in the OUIS. We will also examine the impact of lookahead queries on overall system performance.

8.2.2.2 Interaction with the Data Mining Engine

In addition to the Profiler, the Rule Warehouse also interacts with the Data Mining engine. Each time the mining process runs, the Rule Warehouse must be updated to reflect the changes. These changes primarily impact the `RULES` table, so we focus on the maintenance of this table.

The mining of the session logs can be done with any frequency. The mining process is based on sequential mining as described in Chapter 6. The changes that impact the `RULES` table may include adding new rules, deleting rules that no longer apply, or updating the consequent probabilities for existing rules. We now discuss each type of change in turn. For the purpose of this discussion, we assume that the LD2 design strategy (refer to Figure 21 in Chapter 7) is used.

- **Adding Rules.** When a new rule is generated, it is simply appended to the `RULES` table. This is accomplished by first applying MD5 to obtain the surrogate value for the *RA*. For each of the BDIs (`RA` and `Probability`), the last block is loaded and the data values are appended. For each of the JDIs, a lookup on the referenced BDI is done (requiring a scan of the referenced BDI) to get the pointer value to be stored in the JDI. The last block of the JDI is then loaded, and this pointer value is appended. Thus, the total cost to add a rule is as follows: for each BDI, a read and write access; for each JDI, a scan of the referenced BDI and a read and write access for the JDI.

- **Deleting Rules.** Rules may need to be deleted if they are no longer used. For instance, a change in an e-commerce site may render certain nodes invalid. Structurally, deletion is implemented by maintaining a bit vector, having cardinality equal to the cardinality of the table, where set bits indicate deleted records. There are several ways to determine which rules should be deleted. One such method is to use time, i.e., maintain a last access timestamp for each record. Periodically, a process (similar to the Cache Cleaner in Section 8.2) runs which removes rules that have not been requested within some maximum time threshold. Another method is to explicitly specify a set of rules to delete. This can be done by specifying the set of *RA*s using the SQL `DELETE` command supported by the Rule Warehouse. For either approach, the cost to delete a rule is a scan of the `RA` BDI and a read and write to access the appropriate bit vector block.

- **Updating Rules.** One of the greatest drawbacks of the DataIndex approach is that it is not designed to efficiently support in-place updates. It will typically be quite expensive to load all the blocks that correspond to a particular record. In the case of the Rule Warehouse, however, changes to rules only impact one BDI, the `Probability` BDI. Updates can be made by first retrieving the appropriate `RA` block, which requires a scan of the `RA` BDI. Then the corresponding `Probability` BDI block is loaded and updated. Thus, the total cost to update is a scan of the `RA` BDI plus a read and a write access to the appropriate `Probability` BDI block.

Another approach to maintaining the Rule Warehouse is to rebuild the warehouse each time. However, given the vast size of a typical Rule Warehouse, this will usually be an expensive process. Using the above techniques, it may be necessary to rebuild the warehouse periodically to remove the effects of fragmentation that are the result of record deletions.

Chapter 9

Online User Interaction System Performance Evaluation

In this chapter, we present the results of a set of experiments which demonstrates the performance of the Online User Interaction Server (OUIS), which was described in Chapter 8. These experiments follow the same methodology as those reported in [42]. In fact, some of our results validate the findings in [42]. In our experiments, we focus on the role of the Rule Warehouse and its impact on overall system performance. We first describe our implementation, experimental methodology, and performance metric. Finally, we present our experimental results.

9.1 Implementation Description

The implementation of the OUIS used for these experiments consists primarily of two modules, the *Curio Rule Warehouse* and the *Profiler*. The implementation of Curio was described in detail in Chapter 5, so we do not elaborate on the details here. Both Curio and the Profiler are written in C/C++ using ObjectSpace STL and Communication Toolkits, and compiled with Visual C++ V6.0. Curio and the Profiler each run on separate Pentuim III (450MHz) single-processor NT Server V4.0 (SP5) machines, each with 18GB disk and 256MB RAM. Communication between modules is implemented with sockets over a local Ethernet network.

9.2 Experimental Methodology

Load is simulated on the system by generating user requests, which represent clicks on an e-commerce site. Specifically, a User Process models a specific user's interaction with the site. These User Processes are generated on an NT workstation (SP5) using a user simulator developed in JDK 1.2, and submitted as requests over a local Ethernet network. Each Concurrent User Process (CUP) submits requests as follows: As soon as the CUP enters the system, it submits a request. As soon as it receives a response, it immediately submits a new request, i.e., we do not include "reading time" in our experiments in order to simulate a higher user load. The maximum number of permissible concurrent UPs is denoted as *CUPLevel*. Thus, *CUPLevel* denotes load on the system.

CUPs arrive as a Poisson process with interarrival times averaging *ArrivalRate*, and depart after making *NumClicks* requests to the system, where*NumClicks* is distributed normally with mean *MeanClicks* and standard deviation *StdClicks*. CUPs navigate through the product catalog. Navigation patterns have been studied extensively in the literature [18, 7, 41]. A common finding of these studies is that the number of requests for a given page in a web site most often follows a Zipfian distribution [18, 7]. In the context of web site navigation, this distribution implies that 80% of the users follow 20% of the navigation links. In other words, locality is often present in user navigation. To model such locality in our experiments, each CUP is presented with a set of links from which he can choose his next click, and the link choice is based on a Zipfian distribution drawn from the action probabilities in the rulebase.

The product catalog and rulebase data is simulated. Each product catalog consists of a number of items, *NumItems*, which appear as leaf nodes in the product catalog hierarchy. For each product catalog size, CUP traversals were simulated to generate a ruleset. Each ruleset consists of *RuleBaseSize* rules. The rules in the rulebase and the Zipfian distributions used for CUP navigation during the experiments are both based on the same underlying navigational dataset.

The Profile Cache consists of *CacheSize* cachelines or elements, where *CacheSize* is expressed as *CachePct*, a percentage of the rulebase size. Specifically, *CacheSize*= *CachePct* × *RuleBaseSize*. The Profile Cache is organized on the basis of *Lookahead-Distance* Cache and Data Request Queues.

The rules in the Curio Rule Warehouse are stored according to the logical design $LD1$ described in Section 7.2. As described previously, this schema consists of a fact table and two dimension tables. The query used to retrieve profiles (also described in Section 7.2) is a 3-way join query on the `RULES`, `ACTIONS`, and `NODES` tables with a range restriction on *minProbability* and point restrictions on the `ActionID` and `NodeID` columns. Note that we use a clickstream length (CSL) of 3 for all experiments, so there are 3 action-node column pairs in the `RULES` table.

Table 13 shows the minimum and maximum values of the various system load and data size parameters used in our experiments.

Parameter	**Minimum**	**Maximum**	**Parameter**	**Minimum**	**Maximum**
ArrivalRate	2	2	*NumItems*	5K	60K
MeanClicks	30	30	*RuleBaseSize*	175K	1000K
StdClicks	5	5	*CacheSize*	2K	60K
CUPLevel	10	125	*LookaheadDistance*	1	4

Table 13: Nominal Parameter Values

9.3 Performance Metric

To evaluate the performance of the OUIS system, we use *Average Response Time* (ART), i.e., the average time from a user's click (i.e., request) to the response of the

OUIS. Thus, ART is simply the sum of the response times for all clicks in the system, divided by the total number of clicks made in the system:

$$ART = \frac{\Sigma(t_{response} - t_{click})}{TotalClicks}$$

We chose these timing points, omitting time required for some typical events (e.g., the time required for a web/application server to build a web page for the user) in order to highlight the performance of the OUIS components.

All performance graphs exhibit mean values that have relative half-widths about the mean of less than 10% at the 90% confidence level. We only discuss statistically significant differences in the ensuing reporting section. Each data point represents the average of five experimental values.

9.4 Experimental Results

In this section, we present our experimental results. We first present the baseline results, and then examine the sensitivity of the system performance when certain parameters are varied.

9.4.1 Baseline Results

In the baseline experiments, our aim is to show how DataIndex technology enables true online user interaction. Since virtually all existing personalization technologies use commercial databases for storage and retrieval, we compare the performance of the OUIS under two scenarios: (1) with a DataIndexed Rule Warehouse, and (2) with a widely used commercial data warehousing product. Our contention is that existing commercial solutions are unable to scale to meet the requirements of true online user interaction. We chose Oracle 8*i* as the commercial data warehousing product, since it is a common choice in existing personalization products.

For these experiments, identical datasets corresponding to the LD1 design were loaded into both Curio and Oracle 8*i*. For the Oracle configuration, the physical design of Section 7.3 was used. In other words, bitmapped indexes were defined on the restriction columns (the 3 `Action` and `Node` columns and `Probability` column in the `RULES` table) and on the projection columns (`ACTIONS.Label` and `NODES.Label`). We also assume that bitmapped join indexes (BJIs) are defined on the foreign key columns (`RULES.ActionC` and `RULES.NodeC`).

Oracle was run on a separate machine under Windows NT Server, connected via SQL-Net over a local Ethernet network. All other modules in the system, including the Profiler (for the experiments including the Profiler), were held constant.

Figure 29 shows two curves: (1) Curio with the Profiler (labeled "Curio-Profiler"), and (2) Oracle with the Profiler (Oracle-Profiler), i.e., replacing Curio with Oracle in the OUIS. *CachePct*, *RuleBaseSize*, and LD remain constant at 5%, 500K, and 3, respectively. The points plotted on the curves show steady state values for ART for each *CUPLevel* value.

We first consider the Curio-Profiler curve, starting with a discussion of the general shape of the curve. The curve is exponential, i.e., as *CUPLevel* increases, the rate of increase of the slope increases. At low loads, i.e., between *CUPLevels* of 10 and 50 the slope of the curve is fairly low. In this range, the system is not overwhelmed by requests. As the load on the system increases, the rate of increase of the slope of the curve also increases. Here, a larger number of users in the system leads to an increase in the portion of the rulebase needed to respond to Hint Requests. This, in turn leads to higher cache miss rates and a larger number of requests in the data request queue. A longer data request queue leads to higher response times for cache misses, which leads to higher overall ART values.

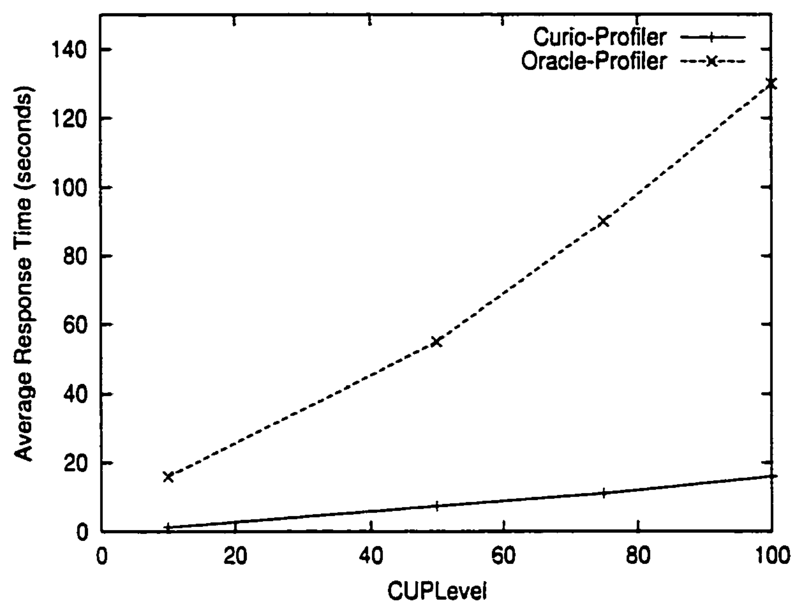


Figure 29: OUIS Performance Using Different Warehousing Technologies

The Oracle-Profiler curve in Figure 29 has the same general shape as the Curio-Profiler curve, i.e., it is exponential. However, the ART values for Oracle-Profiler are much higher than the ART values for Curio-Profiler. In addition, the rate of increase of the slope is much higher for the Oracle-Profiler curve than for the Curio-Profiler curve. These differences are all due to the greater number of disk accesses required to answer the query using the conventional database design. This phenomenon was discussed in detail in Chapters 4 and 7. The greater number of disk accesses in the Oracle case translates into longer response times, causing the data request queue to grow more quickly, and to a larger size. In addition, items are added to the cache more slowly, leading to an even longer data request queue. This leads to the higher overall ARTs, as well as the faster growth of ARTs, as load on the system increases.

Having compared the system performance using different data warehousing technologies for the Rule Warehouse, we move on to discuss our sensitivity experiments.

9.4.2 Sensitivity to Cache Size

In this section, we examine the sensitivity of the OUIS to changes in *CacheSize.* Figure 30[A] shows the impact of having a rule cache. More specifically, Figure 30[A] compares the system performance in the baseline case to the case where there is no rule cache, i.e., *CacheSize*=0. In this figure, the curves labeled "Curio-0" and "Oracle-0" represent the case where there is no rules cache, and the curves labeled "Curio-5" and "Oracle-5" represent the baseline case (these curves are the same as the Curio-Profiler and Oracle-Profiler curves, respectively, in Figure 29). In the case where there is no rules cache, both curves exhibit the same general shape as their baseline counterparts. However, in both cases, performance degrades when no caching is employed. Thus, as expected, the rule cache improves the performance of the Rule Warehouse.

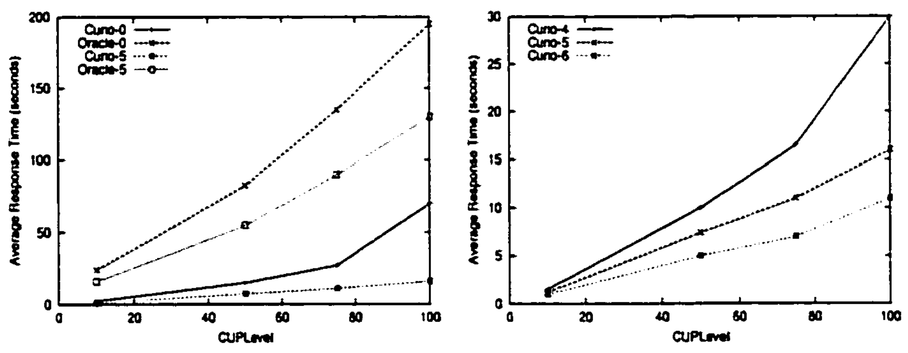


Figure 30: [A] Impact of Rule Cache; [B] Effect of Varying *CacheSize*

As in the baseline case, the Curio-enabled OUIS outperforms the Oracle-enabled OUIS. For instance, when *CUPLevel*=10, the Oracle-enabled OUIS has an ART of about 24 seconds, whereas the Curio-enabled OUIS has an ART of about 2 seconds. When *CUPLevel* is increased to 100, the Oracle-enabled OUIS has an ART of about

195 seconds, whereas the Curio-enabled OUIS has an ART of about 70 seconds. When Curio is used with no caching (Curio-0), the ART starts to increase quickly when *CUPLevel* is increased beyond 75. However, even when Curio is used with no caching, the system performance is significantly better than the case where Oracle is used with caching (Oracle-5).

Figure 30[B] shows the effect of *CacheSize* on the performance of the Curio-enabled OUIS. In this figure, there are three curves for *CachePcts* (i.e., percentages of the *RuleBaseSize*) of 4%, 5%, and 6%, while LD and *RuleBaseSize* remain constant at 3 and 500K, respectively. We note that the curve for *CachePcts*=5% is repeated from Figure 30[A].

The curve for *CachePct*=4% has the same basic shape as the curve for *CachePct*=5%; however, the ART values are higher, and the difference in ART values between *CachePct*=4% and *CachePct*=5% increases as *CUPLevel* increases. This occurs because the smaller cache size causes increased cache contention for *CachePct*=4%. Here, the cache miss rate for *CachePct*=4% is higher than the rate for *CachePct*=5%. This leads to longer data request queues, and higher overall ARTs.

The curve for *CachePct*=6% has the same general shape as the curve for *CachePct*=5%, but has lower ART values. This is due to decreased cache contention – a larger cache retains more useful profiles, leading to fewer requests for the profile warehouse, and shorter data request queues. This, in turn, leads to lower ARTs.

Thus, from Figure 30[B], we conclude that larger cache sizes will lead to lower ARTs, while smaller cache sizes will lead to higher ARTs.

9.4.3 Sensitivity to Warehouse Design

We now examine the impact of the design of the Rule Warehouse on the performance of the overall system. For this experiment, we compare the performance of the system

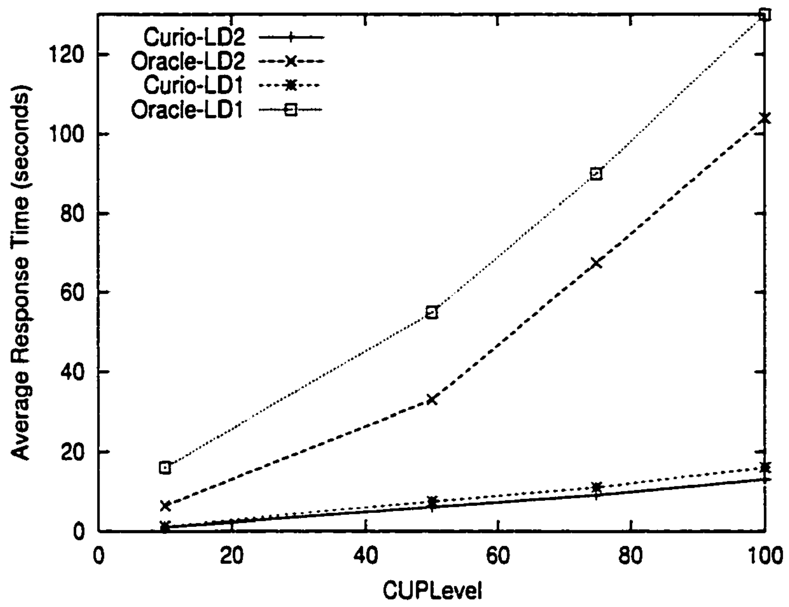


Figure 31: OUIS Performance Using Different Rule Warehouse Designs

(i.e., ART) when the Rule Warehouse design is changed from $LD1$ to $LD2$. The $LD2$ design replaces the 3 `Action-Node` columns in the `RULES` table with a single column (`RA`) containing the MD5 hash value of the RA. MD5 was incorporated into the OUIS using a C++ implementation that is available in the public domain [1].

Figure 31 displays the ART curves for the OUIS using the two different design strategies. Note that the curves labeled "Oracle-LD1" and "Curio-LD1" correspond to the curves labeled "Oracle-Profiler" and Curio-Profiler", respectively, in Figure 29. The overall shape of the curves for LD2 remains the same as for LD1, and both the Curio and Oracle cases benefit from the design change. As the figure shows, the Oracle-Profiler case is much more sensitive to this design change than the Curio-Profiler case. This phenomenon validates our analytical findings in Section 7.4.2.

Our experiments thus far have demonstrated the superior scalability of the Curio-enabled OUIS over the Oracle-enabled OUIS. For this reason, in the remaining experiments, we only consider the Curio-enabled OUIS.

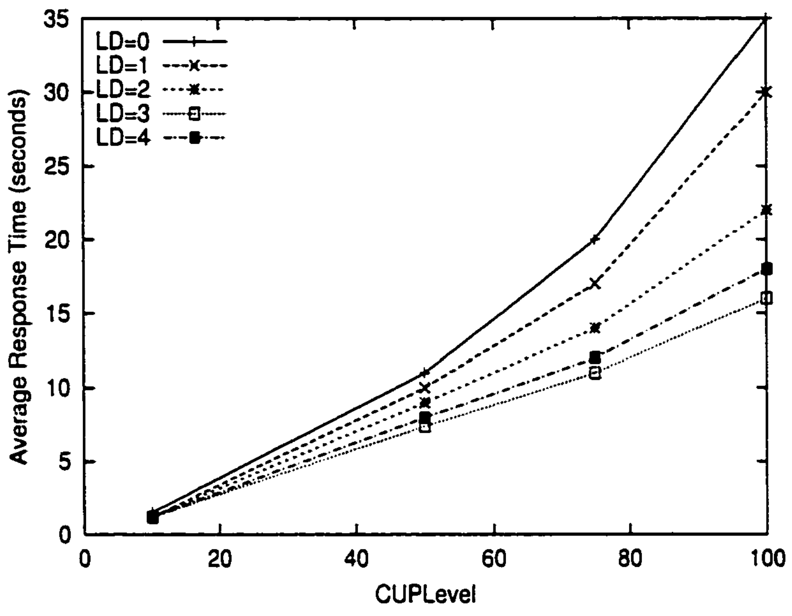


Figure 32: Effect of Varying *Lookahead Distance*

9.4.4 Sensitivity to Lookahead Distance

In this experiment, we show the effect of prefetching profile data and thus examine the utility of the Rule Warehouse in serving anticipatory requests. Figure 32 shows four curves, each showing the change in ART for different *lookahead distances* (LDs) (see Section 8.2.1.1 for a discussion of LD) as *CUPLevel* increases, while the *CachePct* and the *RuleBaseSize* remain constant at 5% and 500K, respectively.

The curve for LD=3 is exactly the same as the Curio-Profiler curve in Figure 29. The curve for LD=2 has the same shape as the LD=3 curve, but appears above the curve for LD=3. This occurs because the system *caches ahead* further for LD=3 than for LD=2. Thus, needed items are less likely to be in cache for LD=2 than for LD=3. This leads to higher response times for LD=2 than for LD=3. This line of reasoning extends to the curve for LD=1 (which also has a shape similar to that of the LD=3 curve) and explains why the ART values for LD=1 are higher than for LD=2, i.e., as the lookahead distance decreases, ART increases.

The uppermost curve (LD=0) shows the response times when no prefetching is employed. This case clearly performs worse than the cases where prefetching is used, indicating that it is indeed beneficial to prefetch.

The curve for LD=4 has a shape similar to the LD=3 curve, but has higher ART values than the LD=3 curve. This occurs because the Profiler with LD=4 *looks too far ahead*, i.e., replaces cache items that are highly likely to be reused with items that have been fetched in anticipation of their need.

Based on the results of this experiment, we conclude that there exists an optimal LD. Clearly, choosing an LD that caches too far ahead will have an effect similar to that of not caching far enough ahead – both lead to sub-optimal ARTs. These results provide further support for the utility of the Rule Warehouse, indicating that the Rule Warehouse does indeed help improve system performance by serving anticipatory requests.

9.4.5 Sensitivity to Size of Rule Warehouse

Figure 33 shows curves for the change in ART for *RuleBaseSizes* of 175K, 500K, and 1000K rules as *CUPLevel* increases, while the *CachePct*, and LD remain constant at 5% and 3, respectively. We note that the curve for 500K is exactly the same as the Curio-Profiler curve in Figure 29. The curve for 175K rules is similar to the curve for 500K rules, but appears below the curve for 500K rules, i.e, shows lower overall ART values. Clearly, a smaller rulebase size will provide lower data warehouse response times, which results in lower response times for the OUIS.

Similarly, a larger rulebase size of 1000K results in higher warehouse response times. This is why the 1000K curve has a shape similar to the 500K curve, but shows higher ART values than the 500K curve.

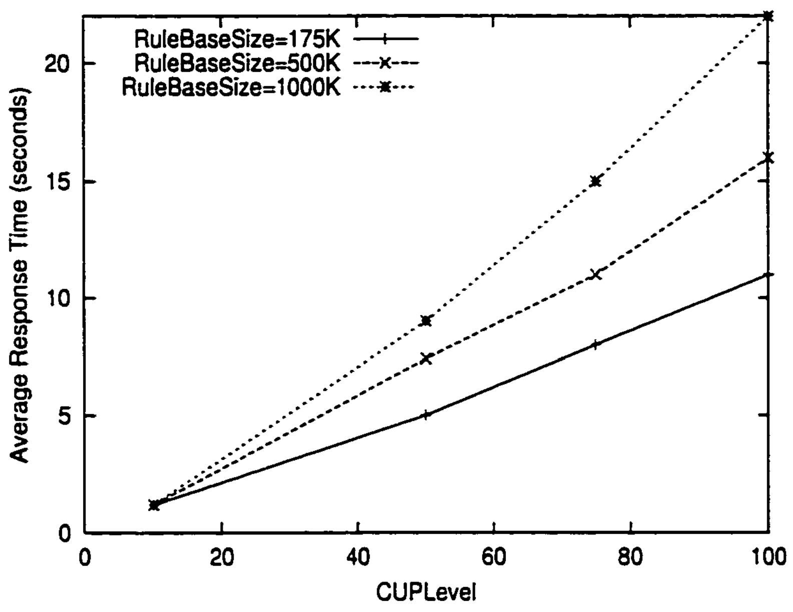


Figure 33: Effect of Varying *RuleBaseSize*

Based on the results of this experiment, we conclude that larger rulebases lead to higher overall ARTs, while smaller rulebases give lower ARTs.

9.5 Discussion

The performance results presented in this section have provided many useful insights regarding the role of the Rule Warehouse in an online user interaction solution. We have shown that the Rule Warehouse plays a key role in such a system. Our results demonstrate the superior scalability of a Rule Warehouse based on the DataIndex paradigm over a warehouse that is based on a conventional design, and thus provides further support for the claim that DataIndex technology enables true online user interaction. We have also examined the sensitivity of the system to changes in certain key parameters (e.g., cache size and rulebase size), and shown that the system scales to support a wide range of the parameters considered in the experiments.

Chapter 10

Discussion

In this dissertation, we have proposed an approach for performing *true* online user interaction. The proposed approach addresses two broad issues: (1) scalable data warehouse design, and (2) efficient generation of online dynamic profiles. We now discuss the implications of our work in both areas.

10.1 Scalable Data Warehouse Design

A key component in any online user interaction solution is a data warehouse. The warehouse stores the historical customer data (e.g., navigational, transactional, demographic) that is used to generate responses. Since responses must be generated in subsecond time frames, the warehouse must be able to provide interactive query response times. There are several factors that make this a difficult task:

- The size of the warehouse is typically vast (e.g., ranging from hundreds of gigabytes to a few terabytes).
- The site will often experience high user loads (e.g., thousands to tens of thousands of simultaneous users at popular sites).
- The queries are typically ad-hoc and complex, involving joins of very large tables.

While there has been significant work in the data warehousing field, existing warehousing solutions are unable to scale to provide interactive query response times under the above-mentioned conditions. In particular, the physical design of existing data warehousing systems mandates storing index structures in addition to the base data. As we have shown in our analyses, this approach often results in significantly higher query processing costs, due to the increased number of disk accesses required to answer queries. To address this issue, we have proposed a new data storage and retrieval paradigm for data warehouses, referred to as *DataIndexes*. With DataIndexes, the structures which store the data also serve as indexes. Hence, there is essentially no indexing cost. We have presented two types of DataIndexes, as well as two efficient algorithms for performing join queries with DataIndexes.

The first algorithm, referred to as *Star Join with Large Memory* (SJL), assumes that enough memory exists to hold the DataIndex structures for all display columns. When this assumption does not hold, a second algorithm, referred to as *Star Join with Small Memory* (SJS), can be used. While SJS is not as efficient as SJL, it has neglible memory requirements.

We have derived expressions that show the expected performance of DataIndexes and a number of other indexing approaches that have been proposed and utilized in data warehousing. Based on these expressions, we have analytically shown that the performance obtainable with DataIndexes is most often superior to that of even the best conventional approach. Specifically, we have analyzed the performance of DataIndexes with respect to *range selections* and *star joins*, two of the most common operations in OLAP. The corresponding results are summarized in Table 14.

We have also presented the results of an implementation based on DataIndexes, which further supports our analytical findings and supports our belief that a DataIndexed data warehouse can indeed provide the subsecond response times required for online user interaction.

Structure	Best for Range Selections When ...
Bitmapped Indexes	Small Ranges
BDI	Medium to Large Ranges and Narrow Columns
JDI	Medium to Large Ranges and Wide Columns
Structure	**Best for Star-Joins When ...**
Bitmapped Indexes & Bitmapped Join Indexes	Very High Selectivities
DataIndexes	Otherwise

Table 14: Summary of Results

In addition to our analytical and implementation results, we summarize a number of other advantages of the DataIndex approach:

Reduced Number of Block Accesses: The vertical partitioning employed by the DataIndex approach brings about an additional advantage. A large portion of OLAP queries access only a subset of the attributes of the tables examined. In a conventional RDWMS, even if the value of a single attribute is needed, entire records must be loaded from disk, because attribute values inside unindexed records are stored contiguously. In other words, conventional RDWMSs most often load extraneous data. Because DataIndexes partition warehouse data by column, only the relevant columns are loaded to answer a query.

High Compressibility: The storage requirements of DataIndexes can be further reduced by compressing each index in a given schema. Though this approach could also potentially be applied to conventional RDWMSs, storage size reductions are likely to be much higher with DataIndexes. Indeed, most compression algorithms will yield higher compression ratios if the set of values being compressed ranges over a small domain (and thus displays repetitions) [50]. This is more likely to happen with a vertical partitioning approach than with a conventional table, where the data is stored row by row. Moreover, while it will sometimes be necessary to decompress the data, there exist query-processing algorithms that can operate on compressed data. We can hence expect that compressed DataIndexes can also provide higher performance gains than compressed versions of the other indexes described in this dissertation.

Better Memory Usage: Queries often access the same column a number of times (e.g., when this column appears both in the `SELECT` and the `WHERE` clause). When these columns are small (which will often be true when compression is used), or the size of main memory is large, query cost can be significantly reduced by keeping one or more of these columns in memory for the duration of the query evaluation. Similarly, since the overall size of the database is smaller with DataIndexes than with conventional structures, it is likely that the number of memory faults will be smaller with DataIndexes. For instance, consider a database that is 700 MB when implemented using DataIndexes and 1.1 GB when implemented using a conventional indexing scheme, such as the B^+-tree . Further assume that the size of main memory is 256 Mbytes. We can then expect the DataIndex approach to find the required column in memory approximately $\frac{256\text{Mbytes}}{700\text{Mbytes}} \approx 36\%$ of the time. On the other hand, the conventional approach will only yield a block hit ratio of about $\frac{256\text{Mbytes}}{1.1\text{Gbytes}} \approx 23\%$. Thus, the DataIndex approach should yield significant performance advantages by reducing the number of times that a data block actually needs to be loaded from disk.

Reduced Memory Requirements: The proposed SJL algorithm has the property that the *memory requirements are independent of the size of the fact table.* Given the extreme size of most warehouses, this is a valuable property, as it allows for the rapid growth that typically occurs in warehouses, particularly those supporting web applications (e.g., clickstream analysis).

Easy Warehouse Refresh: In order to remain up-to-date, a data warehouse must be periodically *refreshed*. That is, its tables and indexes must be updated to reflect changes in the real world. Since a warehouse stores a history, a refresh consists of a series of append operations to the fact table[1] and updates to the corresponding indexes. Although appending data to existing tables can be performed relatively efficiently, updating the corresponding indexes can be quite costly since these indexes might need to be completely reorganized. Since the DataIndex approach does not utilize indexes, it should be clear that updating a DataIndexed data warehouse would not incur this additional index updating cost. The DataIndex refresh process assumes that data is simply appended; i.e., no sorting or other operations are performed. This efficiency in loading was clearly demonstrated in our implementation results in Chapter 5. This property is indeed desirable since, in many application arenas, the time window available for warehouse refreshes is shrinking [20].

In summary, we have proposed efficient data warehouse design techniques. We believe that our proposed techniques are especially applicable in the context of online user interaction.

10.2 Efficient Generation of Online Dynamic Profiles

While the data warehouse provides access to the historical data in an online user interaction solution, it is also necessary to track current user behavior and correlate it with the historical behavior, so that an appropriate response can be generated

[1]Changes to dimension tables occur much less frequently, so we assume that they are not updated during a normal warehouse refresh.

by the application. We have used the term *dynamic profile* to refer to the behavior representation of a user, since it captures changes in user behavior. Generation of dynamic profiles is difficult since these profiles must be generated quickly, even when a site is experiencing heavy user loads.

To address this issue, we have proposed an online user interaction system that consists of a DataIndex-based data warehouse and a caching module. The data warehouse stores the information needed to create dynamic profiles and provides very fast access to this information. The caching module further improves the system performance by storing frequently requested profile information. We have presented an implementation of the proposed online user interaction system, along with a set of performance results which indicate that the system is indeed capable of providing subsecond response times, even under heavy user loads. Our performance results have also demonstrated the importance of an efficient data warehouse in the overall system. In addition, we have shown that the rule warehouse can improve overall system performance by serving anticipatory requests.

10.3 Final Remarks

Our proposed online user interaction system can be readily deployed as part of a typical e-commerce site architecture. There are basically two integration points between our solution and an e-commerce site: (1) session logging, and (2) communication with the web/application server. As mentioned previously, session logging is handled by a client-side Clickstream Monitor (a Java applet) that sends clicks to the server. Communication between the online user interaction solution and the web/application server is needed to pass hint requests and responses, and is handled via the TCP/IP communications protocol.

In this dissertation, we have focused on describing our proposed online user interaction system in the context of electronic commerce. However, our underlying ideas could be applied to virtually any content generation system. For instance, the same basic concepts could be used to deliver targeted content over broadband.

Bibliography

[1] M.T. Abzug. Md5 c++ implementation. http://userpages.umbc.edu/ mabzugl/cs/md5/md5.html.

[2] B. Adelberg, H. Garcia-Molina, and J. Widom. The STRIP rule system for efficiently maintaining derived data. In *Proc. ACM SIGMOD Intl. Conf. on Management of Data*, pages 147–158, Tucson, AZ, May 13-15 1997.

[3] S. Agarwal, R. Agrawal, P.M. Deshpande, A. Gupta, J.F. Naughton, R. Ramakrishnan, and S. Sarawagi. On the computation of multidimensional aggregates. In *Proc. 22nd VLDB Conf.*, Mumbai, India, 1996.

[4] R. Agrawal, A. Gupta, and S. Sarawagi. Modeling multidimensional databases. In *Proc. Thirteenth Intl. Conf. on Data Engineering*, pages 232–243, Birmingham, UK, April 7-11 1997. IEEE.

[5] R. Agrawal, T. Imielinski, and A. Swami. Mining association rules between sets of items in large databases. In *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data*, pages 207–216, May 1993.

[6] R. Agrawal and R. Srikant. Mining sequential patterns. In *Proceedings of the Eleventh International Conference on Data Engineering*, pages 3–14, March 1995.

[7] V. Almeida, A. Bestavros, M. Crovella, and A. de Oliveira. Characterizing reference locality in the www. In *Proceedings of the Fourth International Conference on Parallel and Distributed Information Systems (PDIS '96)*, 1996.

[8] Amazon.com. Amazon.com company information. http://www.amazon.com.

[9] Andromedia. Likeminds. www.andromedia.com/products/likeminds, 1999.

[10] R. Armstrong. Data warehousing: Dealing with the growing pains. In *Proc. Thirteenth Intl. Conf. on Data Engineering*, pages 199–205, Birmingham, UK, April 7-11 1997. IEEE.

[11] G.O. Arocena and A.O. Mendelzon. Web oql: Restructuring documents, databases, and webs. In *Proceedings of the Fourteenth International Conference on Data Engineering*, pages 24–33, February 1998.

[12] P. Atzeni, G. Mecca, and P. Merialdo. Semistructured data in the web: Going back and forth. *SIGMOD Record*, 26(4):16–23, 1997.

[13] T. Barclay, R. Barnes, J. Gray, and P. Sundaresan. Loading databases using dataflow parallelism. *SIGMOD Record*, 23(4), December 1994.

[14] S. Berchtold, D. Keim, and Kriegel H.-P. The x-tree: An index structure for high-dimensional data. In *Proc. 22nd VLDB Conf.*, Mumbai, India, 1996.

[15] Sourav Bhowmick, Wee Keoung Ng, and Sanjay Madria. Schemas for web data: A reverse engineering approach. *Data and Knowledge Engineering*, 39(2):105–142, 2001.

[16] M.W. Blasgen and K.P. Eswaran. On the evaluation of queries in a database system. Technical Report RJ-1745, IBM Corp., San Jose, CA, April 1976.

[17] K. Bratbergsengen. Hashing methods and relational algebra operations. In *Proc. 10th VLDB*, Singapore, August 1984.

[18] L. Breslau, P. Cao, L. Fan, G. Phillips, and S. Shenker. Web caching and zipf-like distributions: evidence and implications. In *Proceedings of INFOCOM'99*. IEEE Press, 2000.

[19] A.G. Buchner, M. Baumgarten, S.S. Anand, M.D. Mulvenna, and J.G. Hughes. Navigation pattern discovery from internet data. In *Proceedings of WE-BKDD'99: Workshop on Web Usage Analysis and User Profiling*, 1999. http://www.acm.org/sigs/sigkdd/proceedings/webkdd99/toconline.htm.

[20] J. Byard and D. Schneider. The ins and outs (and everything in between) of data warehousing. Tutorial in ACM SIGMOD Intl. Conf. on Management of Data, Montreal, Quebec, Canada, June 4-6 1996.

[21] L. Cabibbo and R. Torlone. Querying multidimensional databases. In *Proc. 6th DBPL Workshop*, 1997.

[22] M.J. Carey and D. Kossman. On saying "enough already" in SQL. In *Proc. ACM SIGMOD Intl. Conf. on Management of Data*, pages 219–230, Tucson, AZ, May 13-15 1997.

[23] C.Y. Chan and Y. Ioannidis. Bitmap index design and evaluation. In *Proc. ACM SIGMOD Intl. Conf. on Management of Data*, pages 355–366, Seattle, WA, June 1-4 1998.

[24] D. Chatziantoniou and K.A. Ross. Querying multiple features of groups in relational databases. In *Proc. 22nd VLDB Conf.*, Mumbai, India, 1996.

[25] S. Chaudhuri, R. Krishnamurthy, S. Potamianos, and K. Shim. Optimizing queries with materialized views. In *Proc. 11th ICDE*, pages 190–200, Taipei, Taiwan, March 1995.

[26] S. Chaudhuri and K. Shim. Including group-by in query optimization. In *Proc. 20th Intl. Conf. on Very Large Databases*, pages 131–139, Santiago, Chile, September 1994.

[27] S. Chaudhuri and K. Shim. Optimizing queries with aggregate views. In P. Apers, M. Bouzeghoub, and G. Gardaring, editors, *Advances in Database Technology - EDBT'96 5th Intl. Conf. on Extending Database Technology*, volume 1057 of *Lecture Notes in Computer Science*, pages 167–182. Springer-Verlag, New York, 1996.

[28] S. Chauduri and U. Dayal. An overview of data warehousing and OLAP technology. *SIGMOD Record*, 26(1):65–74, March 1997.

[29] J-H. Chu and G. Knott. An analysis of B-trees and their variants. *Information Systems*, 14(5), 1989.

[30] E. Codd. A relational model for large shared data banks. *Communications of the ACM*, 13(6), June 1970.

[31] L. Colby, T. Griffin, L. Libkin, I. Mumick, and H. Trickey. Algorithms for deferred view maintenance. In *Proc. ACM SIGMOD Intl. Conf. on Management of Data*, Montreal, Quebec, Canada, June 1996.

[32] L.S. Colby and I.S. Mumick. Staggered maintenance of multiple views. In I.S. Mumick and A. Gupta, editors, *Proc. Workshop on Materialized Views: Techniques and Applications*, pages 119–128, Montreal, Canada, June 7 1996.

[33] G. Colliat. OLAP, relational and multidimensional database systems. *SIGMOD Record*, September 1996.

[34] D. Comer. The ubiquitous B-tree. *ACM Computing Surveys*, 11(2):121–138, June 1979.

[35] R. Cooley, B. Mobasher, and J. Srivastava. Data preparation for mining world wide web browsing patterns. *Knowledge and Information Systems*, 1(1):5–32, 1999.

[36] G.P. Copeland and S. Khoshafian. A decomposition storage model. In *Proc. ACM SIGMOD*, pages 268–279, 1985.

[37] International Data Corp. Data warehousing tools: 1998 worldwide markets and trends, report 17622, 1998.

[38] IBM Corporation. Db2 intelligent miner, version 6.1.1. http://www.ibm.com/software/data/iminer.

[39] IBM Corporation. Web sphere commerce suite, version 5.1. http://www.ibm.com/software/webservers/commerce.

[40] PeopleSoft Corporation. Vantive. http://www.peoplesoft.com.

[41] C. Cunha, A. Bestavros, and M. Crovella. Characteristics of www client-based traces. Technical Report 1995-010, Boston University, April 1995.

[42] A. Datta, K. Dutta, D. VanderMeer, K. Ramamritham, and S. Navathe. An architecture to support scalable online personalization on the web. *VLDB Journal*, 10(1):104–117, 2001.

[43] A. Datta, D. VanderMeer, K. Ramamritham, and S. Navathe. Toward a comprehensive model of the content and structure of, and user interaction over, a web site. In *Proceedings of the 2000 Workshop on Technologies for E-Services (TES)*, September 2000.

[44] C. Dyreson. Information retrieval from an incomplete data cube. In *Proc. 22nd VLDB Conf.*, Mumbai, India, 1996.

[45] M. Eisenberg. Calculating the ROI of web-based eCRM initiatives. Presented at BEA eWorld Conference, February 2001.

[46] R. Elmasri and S. Navathe. *Fundamentals of Database Systems.* Addison Wesley, third edition, 2000.

[47] M. Ester, J. Kohlhammer, and H.P. Kriegel. The dc-tree: A fully dynamic index structure for data warehouses. In *Proc. 16th ICDE*, San Diego, CA, 2000.

[48] D. Florescu, A. Levy, D. Suciu, and K. Yagoub. Optimization of run-time management of data intensive web sites. In *Proceedings of the 25th VLDB Conference*, pages 627–638, September 1999.

[49] M. Freeston. A general solution to the n-dimensional B-tree problem. In *Proc. ACM SIGMOD Intl. Conf. on Management of Data*, San Jose, CA, 1995.

[50] C.D. French. Teaching an OLTP database kernel advanced datawarehousing techniques. In *Proc. 13th ICDE*, pages 194–198, Birmingham, UK, April 7-11 1997.

[51] F. Gingras and L.V.S. Lakshmanan. nd-sql: A multi-dimensional language for interoperability and olap. In *Proc. 24rd VLDB Conf.*, New York, August 1998.

[52] P. Goel and B. Iyer. SQL query optimization: Reordering for a general class of queries. In *Proc. ACM SIGMOD Intl. Conf. on Management of Data*, pages 47–56, Montreal, Quebec, Canada, June 4-6 1996.

[53] N. Good, B. Schafer, J. Konstan, A. Borchers, B. Sarwar, J. Herlocker, and J. Riedl. Combining collaborative filtering with personal agents for better recommendations. In *Proceedings of the AAAI-'99 Conference*, pages 439–446, 1999.

[54] G. Graefe. Query evaluation techniques for large databases. *ACM Computing Surveys*, 25(2):73–170, June 1993.

[55] J. Gray, A. Bosworth, A. Layman, and H. Pirahesh. Data cube: A relational aggregation operator generalizing group-by, cross-tab, and sub-totals. Technical Report MSR-TR-95-22, Microsoft Corp., Redmond, WA, July 1995.

[56] T. Griffin and L. Libkin. Incremental maintenance of views with duplicates. In *Proc. ACM SIGMOD Intl. Conf. on Management of Data*, San Jose, CA, May 23-25 1995.

[57] A. Gupta, V. Harinarayan, and D. Quass. Aggregate-query processing in data warehousing environments. In *Proc. 21st VLDB Conf.*, Zurich, Switzerland, 1995.

[58] A. Gupta, V. Harinarayan, A. Rajaraman, and J.D. Ullman. Index selection for OLAP. In *Proc. Thirteenth Intl. Conf. on Data Engineering*, pages 208–219, Birmingham, UK, April 7-11 1997. IEEE.

[59] A. Gupta, H. Jagadish, and I. Mumick. Data integration using self-maintainable views. In *Proc. Fifth Intl. Conf. on Extending Database Technology*, Avignon, France, March 1996.

[60] A. Gupta, I. Mumick, and V. Subrahmanian. Maintaining views incrementally. In *Proc. ACM SIGMOD Intl. Conf. on Management of Data*, pages 26–28, Washington, DC, May 26-28 1993.

[61] A. Guttman. R-trees: a dynamic index structure for spatial searching. In M. Stonebraker, editor, *Readings in Database Systems*, pages 599–609. Morgan Kaufmann Publishers, Inc., San Mateo, CA, 1988.

[62] M. Gyssens and L.V.S. Lakshmanan. A foundation for multi-dimensional databases. In *Proc. 23rd VLDB Conf.*, Athens, Greece, September 1997.

[63] V. Harinarayan, A. Rajaraman, and J.D. Ullman. Implementing data cubes efficiently. In *Proc. ACM SIGMOD*, pages 205–216, Montreal, Canada, June 4-6 1996.

[64] J.M. Hellerstein, P.J. Haas, and H.J. Wang. Online aggregation. In *Proc. ACM SIGMOD Intl. Conf. on Management of Data*, pages 171–182, Tucson, AZ, May 13-15 1997.

[65] J.M. Hellerstein and J.F. Naughton. Query execution techniques for caching expensive methods. In *Proc. ACM SIGMOD Intl. Conf. on Management of Data*, pages 423–424, Montreal, Quebec, Canada, June 1996.

[66] J.L. Hennessy and D.A. Patterson. *Computer Architecture: A Quantitative Approach.* Morgan Kaufmann Publishers, 2nd edition, 1996.

[67] C-T. Ho, R. Agrawal, N. Meggido, and R. Srikant. Range queries in OLAP data cubes. In *Proc. ACM SIGMOD Intl. Conf. on Management of Data*, pages 73–88, Tucson, AZ, May 13-15 1997.

[68] C-T. Ho, J. Bruck, and R. Agrawal. Partial-sum queries in OLAP data cubes using covering codes. In *Proc. 16th ACM Symposium on Principles of Database Systems*, Tucson, AZ, May 1997.

[69] Hyperion Corp. Hyperion essbase olap server. http://www.hyperion.com.

[70] IBM Corp. Informix metacube. http://www.informix.com.

[71] IBM Corp. Db2 universal database version 5.0 for windows nt, 1997.

[72] Informix Software. INFORMIX-online extended parallel server and INFORMIX-universal server: A new generation of decision-support indexing for enterprise data warehouses. White Paper, 1997.

[73] W.H. Inmon. *Building the Data Warehouse.* J. Wiley & Sons, Inc., second edition, 1996.

[74] C. Jermaine, A. Datta, and E. Omiecinski. A novel index supporting high volume data warehouse insertion. In *Proc. 25th VLDB Conf.*, Edinburgh, Scotland, 1999.

[75] S. Khoshafian, G.P. Copeland, T. Jagodis, H. Boral, and P. Valduriez. A query processing strategy for the decomposed storage model. In *Proc. ICDE*, pages 636–643, 1987.

[76] R. Kimball. *The Data Warehouse Toolkit.* J. Wiley & Sons, Inc., first edition, 1996.

[77] R. Kimball and K. Strehlo. Why decision support fails and how to fix it. *SIGMOD Record*, 24(3):92–97, September 1995.

[78] M. Kitsuregawa, M. Nakayama, and M. Takagi. The effect of bucket size tuning in the dynamic hybrid GRACE hash join method. In *Proc. 15th VLDB*, Amsterdam, August 1989.

[79] Y. Kotidis and N. Roussopoulos. Dynamat: A dynamic view management system for data warehouses. In *Proc. ACM SIGMOD Intl. Conf. on Management of Data*, pages 371–382, Philadelphia, Pennsylvania, June 1999.

[80] Wilburt Labio, Ramana Yerneni, and Hector Garcia-Molina. Shrinking the warehouse update window. In *SIGMOD 1999, Proceedings ACM SIGMOD*

International Conference on Management of Data, June 1-3, 1999, Philadephia, Pennsylvania, USA, pages 383–394, 1999.

[81] L.V.S. Lakshmanan, F. Sadri, and I.N. Subramanian. A declarative language for querying and restructuring the web. In *Proceedings of the Sixth International Workshop on Research Issues in Data Engineering - Interopability of Nontraditional Database Systems*, pages 12–21, February 1996.

[82] W. Lehner, J. Albrect, and H. Wedekind. Normal forms for multidimensional databases. In *Proc. 10th Intl. Conf. on Scientific and Statistical Database Management*, pages 63–72, 1998.

[83] Alon Y. Levy, Alberto O. Mendelzon, Yehoshua Sagiv, and Divesh Srivastava. Answering queries using views. In *Proceedings of the Fourteenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, May 22-25, 1995, San Jose, California*, pages 95–104, 1995.

[84] C. Li and X.S. Wang. A data model for supporting on-line analytical processing. In *Proc. Conf. on Information and Knowledge Management*, pages 81–88, Baltimore, MD, November 1996.

[85] D. Lomet, ed. Special issue on materialized views and data warehousing. *IEEE Data Engineering Bulletin*, 18(2), 1995.

[86] Blue Martini. Blue martini personalization. http://www.bluemartini.com.

[87] A.O. Mendelzon, G.A. Mihaila, and T. Milo. Querying the world wide web. *International Journal on Digital Libraries*, 1(1):54–67, 1997.

[88] Microsoft. Asp, c#, vbscript, and asp+. http://www.microsoft.com.

[89] Microsoft Corp. Open Database Connectivity (ODBC). http://www.microsoft.com/data/odbc/.

[90] Microstrategy Corp. Microstrategy intelligence server. http://www.microstrategy.com.

[91] Sun Microsystems. Java server pages. http://java.sun.com/products/jsp/.

[92] I.S. Mumick and A. Gupta, editors. *Proc. Workshop on Materialized Views: Techniques and Applications*, Montreal, Canada, June 7 1996.

[93] I.S. Mumick, D. Quass, and B.S. Mumick. Maintenance of data cubes and summary tables in a warehouse. In *Proc. ACM SIGMOD Intl. Conf. on Management of Data*, pages 100–111, Tucson, AZ, May 13-15 1997.

[94] Nortel Networks. Clarify efrontoffice. http://www.nortelnetworks.com.

[95] J. Nievergelt, H. Hinterberger, and K.C. Sevcik. The grid file: An adaptable, symmetric multikey file structure. In M. Stonebraker, editor, *Readings in Database Systems*, pages 582–598. Morgan-Kaufmann Publishers, Inc., San Mateo, CA, 1988.

[96] P. O'Neil. Model 204 architecture and performance. In *2nd Intl. Workshop on High Performance Transaction Systems (HPTS)*, volume 359 of *Springer-Verlag Lecture Notes on Computer Science*, pages 40–59. Springer-Verlag, Asilomar, CA, 1987.

[97] P. O'Neil and G. Graefe. Multi-table joins through bitmapped join indices. *SIGMOD Record*, 24(3):8–11, September 1995.

[98] P. O'Neil and D. Quass. Improved query performance with variant indexes. In *Proc. ACM SIGMOD Intl. Conf. on Management of Data*, pages 38–49, Tucson, AZ, May 13-15 1997.

[99] Oracle Corporation. Star queries in Oracle8. White Paper, June 1997.

[100] Oracle Corporation. Oracle8 enterprise edition release 8.0.5 for windows nt, 1998.

[101] Oracle Corporation. Oracle 9i Database. http://www.oracle.com/ip/deploy/database/oracle9i/, 2001.

[102] P. Paolini and P. Fraternali. A conceptual model and a tool environment for developing more scalable, dynamic, and customizable web applications. In *Advances in Database Technology - EDBT'98, 6th International Conference on Extending Database Technology*, pages 421–435, March 1998.

[103] D. Peppers and M. Rogers. *The One to One Future: Building Relationships One Customer at a Time.* Bantam Doubleday Dell Publishing, 1997.

[104] Net Perceptions. Net perceptions recommendation engine. www.netperceptions.com, 1999.

[105] B.J. Pine. *Mass Customization.* Harvard Business School Press, 1993.

[106] B.J. Pine and J.H. Gilmore. *The Experience Economy.* Harvard Business School Press, 1999.

[107] D. Quass. Maintenance expressions for views with aggregations. In *Proc. Workshop on Materialized Views: Techniques and Applications*, Montreal, Canada, June 7 1996.

[108] D. Quass and J. Widom. On-line warehouse view maintenance. In *Proc. ACM SIGMOD Intl. Conf. on Management of Data*, pages 393–404, Tucson, AZ, May 13-15 1997.

[109] S.G. Rao, A. Badia, and D. Van Gucht. Providing better support for a class of decision support queries. In *Proc. ACM SIGMOD Intl. Conf. on Management of Data*, pages 217–227, Montreal, Quebec, Canada, June 4-6 1996.

[110] Red Brick Systems. Star schema processing for complex queries. White Paper, July 1997.

[111] Red Brick Systems. Red brick warehouse version 5.1 for windows nt, 1998.

[112] F.F. Reichheld. Loyalty-based management. *Harvard Business School Review*, 2:64–73, 1993.

[113] F.F. Reichheld and W.E. Sasser. Zero defections: quality comes to services. *Harvard Business School Review*, 5:105–7, September-October 1990.

[114] R. Rivest. The MD5 Message-Digest Algorithm. Technical Report RFC-1321, MIT LCS and RSA Data Security, Inc., 1992.

[115] J.T. Robinson. The K-D-B-tree: A search structure for large multi-dimensional dynamic indexes. In *Proc. ACM SIGMOD Intl. Conf. on Management of Data*, pages 10–18, New York, NY, 1981.

[116] N. Roussopoulos, Y. Kotidis, and M. Roussopoulos. Cubetree: Organization of and bulk updates on the data cube. In *Proc. ACM SIGMOD Intl. Conf. on Management of Data*, pages 89–99, Tucson, AZ, May 13-15 1997.

[117] N. Roussopoulos and D. Leifker. Direct spatial search on pictorial databases using packed R-trees. In *Proc. ACM SIGMOD Intl. Conf. on Management of Data*, Austin, TX, 1985.

[118] B. Salzberg. Access methods. *ACM Computing Surveys*, 28(1):117–120, March 1996.

[119] B. Sarwar, G. Karypis, J. Konstan, and J. Riedl. Analysis of recommendation algorithms for e-commerce. In *Proceedings of the 2000 ACM Conference on E-Commerce (EC'00)*, October 2000.

[120] B. Sarwar, J. Konstan, A. Borchers, J. Herlocker, B. Miller, and J. Riedl. Using filtering agents to improve prediction quality in the grouplens research collaborative filtering system. In *Proceedings of the CSCW '98*, 1998.

[121] J.B. Schafer, J. Konstan, and J. Riedl. E-commerce recommendation applications. *Journal of Data Mining and Knowledge Discovery*, 5(1), 2000.

[122] C. Shahabi. Knowledge discovery from users web-page navigation. In *Proceedings of RIDE'97 - Seventh International Workshop on Research Issues in Data Engineering*, 1997.

[123] L.D. Shapiro. Join processing in database systems with large main memories. *ACM TODS*, 11(3), October 1986.

[124] U. Shardanand and P. Maes. Social information filtering: algorithms for automating "word of mouth". In *Conference Proceedings on Human Factors in Computing Systems*, pages 210–217, May 1995.

[125] A. Shoshani. OLAP and statistical databases: Similarities and differences. *ACM TODS*, 22, 1997.

[126] A. Shukla, P.M. Deshpande, J.F. Naughton, and K. Ramasamy. Storage estimation for multidimensional aggregates in the presence of hierarchies. In *Proc. 22nd VLDB Conf.*, Mumbai, India, 1996.

[127] Inc. Siebel Systems. Siebel ebusiness. http://www.siebel.com.

[128] D. Simpson. Corral your storage management costs. *Datamation*, pages 88–93, April 1997.

[129] M. Spiliopoulou, L.S. Faulstich, and K. Winkler. A data miner analyzing the navigational behavior of web users. In *International Conference of ACAI'99: Workshop on Machine Learning in User Modelling*, 1999.

[130] M. Spiliopoulou, M. Hatzopoulos, and Y. Cotronis. Parallel optimization of large join queries with set operators and aggregates in a parallel environment supporting pipeline. *IEEE TKDE*, 8(3):429–45, June 1996.

[131] M. Spilioupoulou, L.C. Faulstich, P. Atzeni, A. Mendelzon, and G. Mecca. Wum: A tool for web utilization analysis. In *International Workshop WebDB'98: World Wide Web and Databases*, pages 184–203, 1998.

[132] R. Srikant and R. Agrawal. Mining sequential patterns: Generalizations and performance improvements. In *Advances in Database Technology - EDBT'96, 5th International Conference on Extending Database Technology*, pages 3–17, March 1996.

[133] Sybase, Inc. Sybase IQ – optimizing interactive performance for the data warehouse. White Paper, 1997.

[134] A.S. Tanenbaum. *Modern Operating Systems.* Prentice Hall, second edition, 2001.

[135] Broadvision Technologies. Broadvision. www.broadvision.com, 1999.

[136] Engage Technologies. Engage e-commerce product suite. www.engage.com, 1999.

[137] H. Thomas and A. Datta. A conceptual model and algebra for on-line analytical processing in decision support databases. *Information Systems Research*, 12(1):83–102, March 2001.

[138] Transaction Processing Performance Council, San Jose, CA. *TPC Benchmark D (Decision Support) Standard Specification*, revision 1.2.3 edition, June 1997.

[139] P. Valduriez. Join indices. *ACM Transactions on Database Systems*, 12(2):218–246, June 1987.

[140] P. Valduriez, S. Khoshafian, and G.P. Copeland. Implementation techniques of complex objects. In *Proc. VLDB*, pages 101–110, 1986.

[141] B. Vance and D. Maier. Rapid bushy join-order optimization with cartesian products. In *Proc. ACM SIGMOD Intl. Conf. on Management of Data*, pages 35–46, Montreal, Quebec, Canada, June 4-6 1996.

[142] D. VanderMeer, K. Dutta, A. Datta, K. Ramamritham, and S. Navathe. Enabling scalable online personalization on the web. In *Proceedings of the 2000 ACM E-Commerce Conference (EC'00)*, pages 185–196, October 2000.

[143] P. Vassiliadis. Modeling multidimensional databases, cubes and cube operations. In *Proc. 10th Intl. Conf. on Scientific and Statistical Database Management*, pages 53–62, 1998.

[144] I.R. Viguier, A. Datta, and K. Ramamritham. Exact performance expressions for olap queries. Technical Report GOOD-TR-9709, U. of Arizona, 1997. Available from `http://loochi.bpa.arizona.edu`.

[145] S.V. Vrbsky and J.W.S. Liu. APPROXIMATE – a query processor that produces monotonically improving approximate answers. *IEEE Transactions on Knowledge and Data Engineering*, 5(6):1056–1068, 1993.

[146] M-C. Wu and A. Buchmann. Encoded bitmap indexing for data warehouses. In *Proc. 14th ICDE*, pages 220–230, Orlando, Florida, February 1998.

[147] Yongqiao Xiao and Margaret Dunham. Efficient mining of traversal patterns. *Data and Knowledge Engineering*, 39(2):191–214, 2001.

[148] W.P. Yan and P.A. Larson. Eager aggregation and lazy aggregation. In *Proc. 21st Intl. Conf. on Very Large Databases*, pages 345–357, Zurich, Switzerland, September 1995.

[149] Jun Yang and Jennifer Widom. Temporal view self-maintenance. In *Advances in Database Technology - EDBT 2000, 7th International Conference on Extending Database Technology, Konstanz, Germany, March 27-31, 2000, Proceedings*, volume 1777 of *Lecture Notes in Computer Science*, pages 395–412. Springer, 2000.

[150] Y. Zhao, P.M. Deshpande, and J.F. Naughton. An array-based algorithm for simultaneous multidimensional aggregates. In *Proc. ACM SIGMOD Intl. Conf. on Management of Data*, pages 159–170, Tucson, AZ, May 13-15 1997.

[151] Y. Zhuge, H. Garcia-Molina, J. Hammer, and J. Widom. View maintenance in a warehousing environment. In *Proc. ACM SIGMOD Intl. Conf. on Management of Data*, pages 316–327, San Jose, CA, May 23-25 1995.