A

Query Optimization in Temporal Relational Databases

By

Samir Adel Mohammad

Supervisor

Assoc. Prof. Munib Qutaishat

Submitted in Partial Fulfillment of the Requirements for the

Degree of Master of Science in

Computer Science

Faculty of Graduate Studies

University of Jordan

May 2002

This thesis was successfully defended and approved on

……..

Examining Committee	**Signature**
Dr. Munib Qutaishat / Chairperson Assoc. Prof. of Database	……
Dr. Saleh Oqeili / Member Prof. of Computer Architecture	.....
Dr. Riad Jabri / Member Assoc. Prof. of Programming Languages	.....
Dr. Khalil El-Hindi / Member Asst. Prof. of Artificial Intelligence	.....
Dr. Rehab Duwairi / Member Asst. Prof. of Database	.....

Dedication

To my father,

To my wife, and

To Mrs. Beth Kuttab,

For support and understanding

Acknowledgements

I would like to thank Dr. M. Qutaishat for his guidance and support and the examination committee for their presence at my defense.

Also, many thanks to all staff members of the Department of Computer Science at the University of Jordan for all the learning and enlightenment they have given me during the past three years of my graduate studies. I wish all staff members and the department a continuous advancement.

List of Contents

List of Tables

List of Figures

Appendices

M

Query Optimization in Temporal Relational Databases

By

Samir Adel Mohammad

Supervisor

Assoc. Prof. Munib Qutaishat

Abstract

Temporal databases are repositories of time dependent information. The main difference from standard database (relational database) systems, is the need to possibly store a limitless number of tuples that grows over time.

Many proposals have been introduced to establish models that incorporate time dimension into standard relational databases. Some of these models suggest using attribute time-stamps. While other models suggest using single relation time-stamps, or multiple relations time stamps.

This thesis try to address several issues connected to multiple relations time-stamps temporal relational model and the development of temporal relational databases. It discusses different temporal relational database models. Determine the storage requirements for multiple relations temporal relational model and single relation temporal relational model. And comparing storage costs for multiple relations temporal relational model with single relation temporal relational model. Also, new hash-clustered indexing structure has been designed to accommodate efficient access for tuples that are indexed on time-stamps. In temporal schema and temporal index sections, we demonstrate that our solutions are practical by outlining efficient implementations. Furthermore, new time intersection equi-join algorithms have been written. These algorithms have been designed to handle special types of temporal relations, such like continuous and event dependents temporal relations.

1. INTRODUCTION

1.1 Overview

In the rest of this section we will discuss temporal databases in general, query optimization, and illustrate the guidance through which this thesis is motivated by.

In the next section, query optimization methodology in classical relational database will be outlined. Then discussion of query optimization in temporal relational databases will be carried out in the rest of the thesis. Three major areas in query optimization will be discussed. In section 3.1 temporal relational database schema will be discussed. In section 3.2 temporal relational database index will be discussed. And in section 3.3 joining in temporal relational database will be discussed. In the last section a summary of findings and a brief for the future researches with regards to the findings will be outlined.

1.2 Temporal Relational Database

Temporal databases are becoming very popular lately. Many applications depend on the temporal aspect in their usage such as hotel reservation,

medical services, insurance companies, travel agencies, communications, and many other applications. Importance of temporal databases arises from the fact that they are always occupying a very huge amount of memory space. All transactions are recorded all the times. In contrary to transaction databases, only current status of an attribute is recorded. This kind of database that holds only the current status is called snapshot database. Snapshot relations that are the core of snapshot database constitutes a small fraction of temporal relational database.

Therefore, required memory space for temporal relational databases are larger than the required memory space that is required for regular classical databases, because of the reason stated in last paragraph. Plus the fact that each tuple in temporal relational databases is associated with 2-4 time-stamps. Because of these reasons, query optimization techniques becomes more essential for temporal database, more than they are essential for classical databases, in order to deal with this enormous quantity of data. Another important fact that explains the need for query optimization in relational temporal database is that the related joins between different relations are always of inequality join type. Inequality joins requires space and processing time in temporal database order of magnitude greater than the needed space and time for equality joins which is popular in regular classical relational database.

As temporal database area is an exciting research area, many researches have been conducted in this field generally. 1200 papers on temporal databases have appeared Up to year 1996. Over 300 papers during 1995-1996 alone [Tsotras, 1996].

1.3 Temporal Vs. Classical Relational Databases

The importance of temporal relational database intensive researches comes from the following facts:

- Temporal databases are always very huge in size. Whereas classical databases size is small if it compare to temporal database.
- Joins in regular databases are of equality types. But joins in temporal relations are of in-equality types.
- Researches of algorithm complexity for regular relational database are fixed and well known. But in temporal database it varies depending on the used infrastructure schemes, such like schemes that use attribute time stamping, single tuple time stamps, multiple tuples time stamps and many more.
- Agreements on universal relational database definitions have been achieved. But for temporal database there are many to choose from, with regard to temporal relational databases. Besides object oriented approaches.

- A comprehensive solid foundation discipline, so to speak, has emerged to optimize query in classical relational database. Steps are well identified and commercial query optimizers are available. In contrast, the picture for temporal relational databases is still ambiguous and prototypes for temporal query optimizers are still under testing.
- Index in regular databases involves certain key/keys, but in temporal relational database there are four keys and they are all as important as each other's are.

  Many researchers have considered in their researches the possibility of using the existing relational query languages in building a temporal query language in upward compatibility [Bair, 1997]. Others have designed a temporal middleware to optimize temporal queries using conventional DBMS [Slivinskas, 2001]. Also Snodgrass [1996A] and [1997B] had integrated valid time and transaction time into SQL query language. Regular SQL query language is illustrated in Qutaishat [1999]. These kinds of approaches in not preferred in temporal database implementation. Simulating temporal database by using conventional database systems will limit our capabilities to those features that can be supported by classical database.

  As an example:

- We will not be able to use TSQL or TQUEL. Complete survey for temporal query languages can be found in Chomicki [1995] and Toman [1995].
- We will not be able to use special constructs for attribute characteristics, such like Granularity, Data points, Event Points, Regularity, Change Points, and Type.
- We will not be able to use constructs for Associations between Temporal Attributes.
- Classical DBMS only support functionality to access a single state of the real world, usually the most recent one. Temporal DBMS needs to have access not only to the most recent state, but also to past and even future states.

  Furthermore, temporal manner and temporal nature in a database are fundamentally distinct and possess features that are better be dealt with through a new view. This view has to deal efficiently with temporal tuples that contains surrogate attribute, temporal attribute, start time, and end time.

1.4 Importance of Query Optimization

In database management systems, we do not expect users to write their queries in a way that suggests the most efficient evaluation plan. Rather, it

is the responsibility of the system to construct a query-evaluation plan that minimizes the cost of query evaluation. The most relevant performance measure in query optimization is usually the number of disk accesses [Silberschatz, 1997] in regular database. Whereas, it is the communication cost in distributed systems [Ozsu, 1999].

Query optimization is the process of selecting the most efficient query-evaluation plan for a query. One aspect of optimization occurs at the relational-algebra level. At this level, an attempt is made to find an expression that is equivalent to the given expression, but that is more efficient to execute. The other aspect involves the selection of a detailed strategy for processing the query, such as choosing the algorithm to use for executing an operation, choosing the specific indexes to use, and so on.

To choose among different query-evaluation plans, the optimizer has to estimate the cost of each evaluation plan. Computing the precise cost of evaluation plan is usually not possible without actually evaluating the plan. Instead, optimizers make use of statistical information about the relations, such as relation sizes and index depths, to make a good estimate of the cost of a plan.

The optimizer estimates the cost of the different evaluation plans. If an index is available on a joining attribute, then the evaluation plan in which

the selection is done using the index, is likely to have the lowest cost, and thus, to be chosen.

Once the query plan is chosen, the query is evaluated with that plan, and the result of the query is outputted.

1.5 Basic Areas of Query Optimization

Three major areas will be discussed in temporal relational query optimization. These three areas are Temporal Relational Database Schemes, Temporal Relational Database Indexes, and Temporal Relational Database Joins. They will be discussed in sections 3.1, 3.2, and 3.3, respectively. These areas are strongly related to each other. Also, they are dependent on each other. Elaboration to the proposals in one area can not be carried out properly without providing an adequate foundation in the other prerequisite areas. These areas form the core parts of query optimization in relational databases as well as in temporal relational databases. Most research papers are concentrating in these areas [Tansel, 1993]. These areas will be discussed and enriched with new techniques and/or with some suggested improvements to the existing ones.

Database schema and structure is very important. It will be seen in section 3.1 how different schema affects remarkably the storage costs.

Indexes have a magic effect in query performance. An example will be illustrated in section 2 how indexes affect performance. This example was quoted from Ioannidis [1995]. In this example we use a flat query that join two relations that have indexes. Three plans were proposed to solve the query. The first does not use any available indexes for any of the two relations. The second plan uses one index for only one relation. The third plan uses both relations' indexes. The first plan took more than 24 hours to finish, the second plan took almost an hour, and the third plan took a fraction of a second. The execution time has dropped from more than 24 hours to less than one second with the help of indexes. That's why we have included a section in our thesis for indexes.

Also, indexing alone is not efficient if the proper method for implementing the join is unknown. Hence, a section has been included in this thesis to discuss joins in temporal relational database.

Considerably, the accomplishment of DBMS is measured by the sophistication and distinction of its query optimizer.

Although query optimization exists as a field for more than twenty years, it is very surprising how fresh it remains in terms of being a source of research problems [Ioannidis, 1995].

2. LITERATURE REVIEW FOR QUERY OPTIMIZATION IN RELATIONAL DATABASES

2.1 Introduction

For any given query, the operating DBMS will have many plans to process the query. And the question that is raised here is which plan should the DBMS follow? Even though, all plans will produce the same desired output. But the cheapest plan should be followed. The cost could be measured in time and resources.

Memory is an important factor in temporal query optimization. Because a large amount of data needs to be processed, more used memory will improve the executing time. The needed time to produce a query result is dictated by disk access and processing time. But since processing time is too small to be considered, then number of disk access will be the main factor in query optimization.

In large databases, the cost of processing a query is usually dominated by disk access [Silberschatz, 1997]. It is measured by the number of block transfers from disk, which is slow compared to memory access. Thus, strategy that should be followed in order to process a query should require minimum number of disk accesses. The optimizer should be smart enough

to decide which strategy is best. Also, time spent by the optimizer to choose a strategy should not exceed the saved time by choosing such best and most efficient strategy. In distributed systems, the communication time is generally the dominant factor in measuring the cost [Ozsu, 1999]. The difference in a query processing time for a given query could be enormous. For that reason query optimization became necessary in DBMSs. As an example quoted from Ioannidis [1995]. Consider the following database schema:

Employee (Ename, Eage, Esal, Edno)

Department (Dno. Dname, Dfloor, Dbudget)

Consider the following SQL query:

Select Ename, Dfloor

From Employee, Department

Where Employee.Edno=Department.Dno and Esal > 50,000

Assume the following characteristics applies for the used database schema:

Pages # (Employee relation)	:	20,000
Pages # (Department relation)	:	10
Tuples # (Employee relation)	:	100,000
Tuples # (Department relation)	:	100

Indices (Employee relation) : Clustered B+ tree on Esal key,
(3-Levels deep)

Indices (Department relation) : Clustered hashing on Dno key,
(1.2 pages of Avg. bucket length)

Buffer pages # : 3

Disk page access cost : 20 ms

According to the given assumptions about the database schema. The above query could be solved in too many ways. But in order to illustrate the enormous difference in processing time, let us consider the following three plans:

Plan 1 : Use the B+ index in Employee to extract records that have Esal > 50,000 then use the hash index in Department to extract records for the corresponding Edno.

Plan 2 : For each tuple in Department relation scan all Employee relation to search for matching Edno and Esal > 50,000.

Plan 3 : Cross product Department relation with Employee relation, then scan the result to search for records that have Edno=Dno and Esal > 50,000.

Calculating the I/O cost of these three plans according to the above parameters will produce the following results. The time cost for plan 1 is

0.32 seconds. For plan 2, the cost is more than an hour. And for plan 3 the cost is more than 24 hours. This example illustrates the needs to have an optimizer in the operating DBMSs in order to choose the cheapest alternative. Note that two indexes have been used in plan 1.

The stages that a query runs through in a DBMS in order to produce an answer for a query is depicted in Figure 2.1.

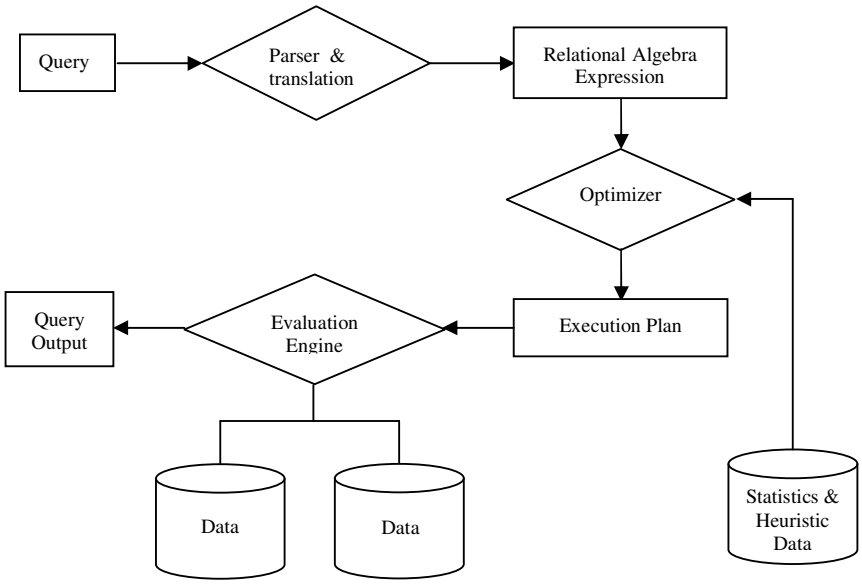


Figure 2.1. Optimization flow in DBMS

As can be seen from Figure 2.1 . The basic steps are:

1. Parsing and translation.
2. Optimization.
3. Evaluation.

The query parser checks the validity of the query syntax and translates it into some kind of an internal form that is understandable by the system. It

verifies that the relation names appearing in the query are names of relations in the database [Aho *et al*., 1986] .

The optimizer generates all the possible equivalent algebraic forms that are qualified for further investigation of the cost. Once the optimizer decides on the cheapest algebraic form, it passes that form to the evaluation engine. The evaluation engine generates the code for that algebraic form and then proceeds in query processing to generate the query result.

There are two types of queries, they are embedded queries and interactive (Ad-Hoc) queries. The embedded query will go through the above 3 stages depicted is Figure 2.1 only for the first time. Afterwards, if it is called a second time, it will skip the optimization step and go directly to step 3 (query evaluation). Whereas for interactive ad-hoc queries, all 3 stages are passed through every time a query is activated. In embedded queries, the query is translated into a reusable code during compiling time, and can be used again repeatedly during run time. In case of embedded queries, an exhaustive search approach is often used where almost all possible execution strategies are considered [Selinger *et al*., 1979]. Optimization done through these two types of queries is basically the same. The embedded queries have a shortcoming, such that the given information at the execution run time may be different than

information that was available at the compiling time. Such like: existence of indexes and size of buffer.

In the following section, the architecture of query optimizer and its components will be illustrated. In the sections following the next section, a more deeply investigation of these components will be carried out.

2.2 Architecture of Query Optimizer

The main objective for a query optimizer is to find the most efficient plan to execute a query correctly. When a query is received from the parser it passes through many stages that all together compose the optimizer. These stages are illustrated in Figure 2.2.

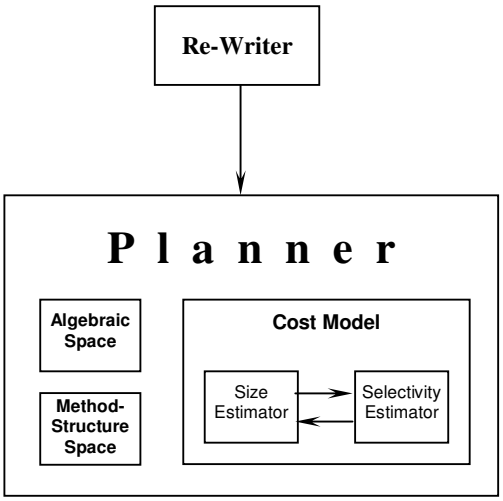


Figure 2.2. Query optimizer architecture

As can be seen from Figure 2.2 . The optimizer consists of main two components, Re-Writer Model, and Planner Model. The Planner model control and collects data from three models, Algebraic Space, Method-Structure Space, and Cost Model.

The cost model estimates the cost for estimated size and estimated selectivity as per received plan from the planner through the size and selectivity estimators. The re-writer model rewrites the query in more efficient clearer way if necessary. As an example, replacing views with their exact naming, and flattening out of nested queries [Kim, 1982]. This stage is identified as declarative stage because it acts on static characteristics of the database schema.

The planner is the most important model in the optimizer. Among all the possible plans to solve the query, the planner is responsible for choosing the most efficient (cheapest) plan to process the query. It does that by limiting the space of execution plans. This space is determined with the help of two models, algebraic space and method-structure space. The options that are limited by searching space are tested afterwards by the cost model to determine the cheapest.

Algebraic basis provided by relational models is of considerable help in query optimization. Algebraic space determines the primarily cheapest sets of actions that are to be executed in order to generate the result for

the given query. These sets of actions are represented in relational algebra formulas or as a tree form. Once the strongest candidates' sets of actions are determined by algebraic space, the method-structure space is used to indicate the methods at which each set can be implemented. These methods are determined by using nested loops, merge scan, or hash join. According to the available supporting data structure and indexes.

Cost model determines the cost of each implementation plan. Each implementation plan' cost will vary depending on the used join method, index type, and order of execution steps [Ioannidis, 1990]. Many factors may affect the cost such like the buffer size, disk-CPU overlap, and random or sequential access to the records.

The more available buffer size, the more temporal data that would be processed at one time, which will lead to less execution time.

If the disk-CPU overlap is synchronized then efficiency will be improved. Synchronization can be established by having the processor processing a chunk of data while the I/O controller is extracting a chunk of data from the hard disk. By the time the CPU finishes processing the chunk it has, the I/O controller will finish extracting a chunk of data from the hard disk and passes it to the CPU at the same time. And so on. Therefore, the CPU will be kept busy most of the time, which will lead to improvement in efficiency.

Hard disk access is associated with disk access time and disk latency time. To transfer a disk block, given its address, the disk controller must first mechanically position the read/write head on the correct track. The time required to do this is called the seek time. Following that, there is another delay called the rotational delay or latency, that is the time it takes the beginning of the desired block to rotates into position under the read/write head.

Two sub-models are used within cost model. The first is the size estimator, which is responsible for determining the size of used relations and indexes. Also responsible for estimating the size of intermediate relations if it's applicable. The second model is the selectivity estimator, which is responsible for finding the distribution of an attribute values within a relation. The cost model uses statistics that are saved in the database catalogue. [Ozkaraham, 1990] have illustrated query optimization in regular relational databases clearly.

In the following sections, these models will be discussed in more details.

2.3 The Re-Writer

The re-writer model duty are mainly re-writing queries in an equivalent manner to the original query, after replacing views with it's related relations and flattening out the nested queries. A new area of re-writing

has been explored, and that is semantic optimization re-writing. This approach can be applied to temporal relational database efficiently. As an example: Suppose that a relation that holds transaction time for the employees in a firm is known to start at 1/1/1997 and ends at the current time. If we get a query that is inquiring about employees before or after that time interval, it will be rejected or notified of this fact.

As another example: Suppose there is a DBMS for an airlines company. Assume that the business roles includes the following two roles :

- No Airplane captain can fly for more that 30 hrs per week.
- No flight steward is allowed to fly more than 50 hrs per week.

Assume that a primary index sorted on hiring dates exists in the database. For the following query:

Select Emp

From Employee, Fly_Schedule

Where Employee.Rank= Steward and

Sum(Fly_Schedule.FlyHrs) > 10,000

Instead of summing up all flying hours for all stewards. And since we know that no steward can fly for more than 50 Hrs a week, and 10,000/50=200. Then we know that we have to search for stewards that are hired before 200 weeks ago and that is 3.7 years from now. Suppose the date was then June 1, 1995.

If the optimizer is equipped with a semantic re-writer [King, 1981] and [Siegel *et al*., 1992], in which this semantic re-writer is also equipped with heuristic temporal engine. Then this engine may deduct a statement from the previous paragraph and augment this statement as an extra statement to the original query (third line in the query). Therefore, the new query will be as suggested below:

Select Emp

From Employee, Fly_Schedule

Where Employee.StartDate<= #1/6/1995# and

Employee.Rank = Steward and

Sum(Fly_Shedule.FlyHrs)>10,000

Since we have primary index on "StartDate" key field in the Employee relation, then enormous time can be saved.

Based on the previous examples, we could embed to the optimizer in temporal relational database a specialized heuristic engine that is capable of enforcing temporal constraints in data entry. Also, this engine should be able to infer shortest way to re-write a query in order to execute it in less time, in accordance with the business rules. So the optimizer architecture depicted in Figure 2.2 , may be enhanced as in Figure 2.3.

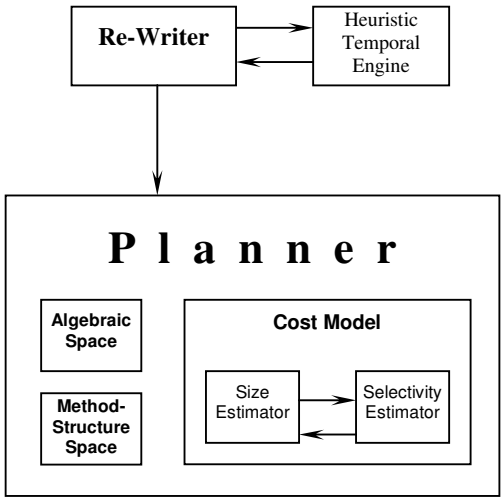


Figure 2.3. Enhanced temporal relational database optimizer architecture

The re-writing stage is identified as declarative stage because it acts upon static characteristics of the database schema.

2.4 Algebraic Space

As explained above, Algebraic space determines the primarily cheapest sets of actions that would be executed in order to generate the result for the given query. These sets of actions are represented in relational algebra formulas or as a tree form. In the following two subsection we will review "relational algebra transformation rules" and "query trees".

2.4.1 Relational algebra transformation rules

The main role of query processor is to choose the best relational algebra query among all equivalent ones. As number of queries increases in the

query, the query may be represented by very large number of equivalent queries [Ibaraki, 1984], which is reduced to the strongest candidate equivalent relational algebra unit. Many rules can be implemented to transform a query into another equivalent one that is cheaper or could be further simplified to be cheaper. These rules are as follows:

1- Cascade of π: For a cascaded π operation. All π can be ignored except the last one, since it includes all the cascaded ones

$\pi_{List1}(\pi_{List2}(\ldots (\pi_{Listn}(R)) \ldots)) = \pi_{List1}(R)$

2- Cascade of σ: A sequence of cascaded σ operations according to some conditions c1, c2, ... , cn in the same relation, can be conjunct in a single σ operation with conjunctive conditions c1, c2, ... , cn.

$\sigma_{c1}(\sigma_{c2}(\ldots(\sigma_{cn}(R))\ldots)) = \sigma_{c1 \text{ and } c2 \text{ and } \ldots \text{ and } cn}(R)$

3- Commutativity of σ:

$\sigma_{c1}(\sigma_{c2}(R)) = \sigma_{c2}(\sigma_{c1}(R))$

4- Commutativity of $\bowtie_c$ and $\bowtie$.

$R \bowtie_c S = S \bowtie_c R$

$R \bowtie S = S \bowtie R$

5- Commuting σ with π: Only if the selection condition is related to the projected attributes A1, A2, ... , An.

$\pi_{A1,A2,\ldots,An}(\sigma_c(R)) = \sigma_c(\pi_{A1,A2,\ldots,An}(R))$

6- Commuting σ with $\bowtie_c$ (or $\bowtie$)

6.1 : If the selection condition attributes are located in R then:

$\sigma_c(R \bowtie S) = (\sigma_c(R)) \bowtie S$

6.2 : If the selection condition in C1 is related to R and selection condition in C2 is related to S and condition c = c1 and c2 then:

$(\sigma_{c1}(R)) \bowtie_{c2} (S)) = \sigma_c(R \bowtie_c S)$

6.3 : The same rules is applicable for 6.1 and 6.2 if we replace $\bowtie$ with $\bowtie_c$

7- Commuting π with $\bowtie_c$ (or $\bowtie$)

7.1 : Suppose there is a set of attributes L={A1, A2, ... , An, B1, B2, ... , Bn), where A1, A2, ... , An belongs to relation R and B1, B2, ... , Bn belongs to relation S. If the join condition requires only attributes that exist in L then:

$\pi_L(R \bowtie_c S) = (\pi_{A1,\ldots,An}(R)) \bowtie_c (\pi_{B1,\ldots,Bn}(S))$

7.2 : If all assumptions in 7.1 are applicable, and there are { An+1, ... , An+k } and { Bn+1, ... , Bn+k}attribute in R and S, respectively, that are related to the condition c and does not exists in L then:

$\bowtie$ $\bowtie$

$\pi_L(R \bowtie_c S) = \pi_L [(\pi_{A1, \ldots, An, An+1, \ldots, An+k} (R)) \bowtie_c (\pi_{B1, \ldots, Bn, Bn+1, \ldots, Bn+k} (S))]$

7.3 : The same rules is applicable for 7.1 if we replace $\bowtie_c$ with $\times$, and it is not applicable for 7.2 because of the presence of condition attributes.

8- Associativity of $\bowtie$, $\bowtie_c$, $\times$, $\cup$, and $\cap$: the symbol ϕ may be substituted with any of these four operators:

$(R \phi S) \phi T) = R \phi (S \phi T)$

9- Commutativity of $\cap$ and $\cup$: the symbol ϕ may be substituted with any of these two operators:

$R \phi S = S \phi R$

10- Commuting π with $\cup$:

$\pi_L (R \cup S) = (\pi_L (R)) \cup (\pi_L (S))$

11- Commuting σ with $\cup$, $\cap$, and $-$: the symbol ϕ may be substituted with any of these three operators

$\sigma_c (R \phi S) = (\sigma_c (R)) \phi (\sigma_c (S))$

12- Change σ and $\times$ into $\bowtie_c$: If the operator $\times$ is followed by the operator σ then replace both with $\bowtie_c$ as follows:

$\sigma_c (R \times S) = R \bowtie_c S$

13- The transformation using logical operations $\wedge$, $\vee$, and $\neg$:

13.1 $P_1 \wedge P_2 \Leftrightarrow P_2 \wedge P_1$

13.2 $P_1 \vee P_2 \Leftrightarrow P_2 \vee P_1$

13.3 $P_1 \wedge (P_2 \wedge P_3) \Leftrightarrow (P_1 \wedge P_2) \wedge P_3$

13.4 $P_1 \vee (P_2 \vee P_3) \Leftrightarrow (P_1 \vee P_2) \vee P$

13.5 $P_1 \wedge (P_2 \vee P_3) \Leftrightarrow (P_1 \wedge P_2) \vee (P_1 \wedge P3)$

13.6 $P_1 \vee (P_2 \wedge P_3) \Leftrightarrow (P_1 \vee P_2) \wedge (P_1 \vee P3)$

13.7 $\neg (P_2 \wedge P_3) \Leftrightarrow \neg P_1 \vee \neg P_2$

13.8 $\neg (P_2 \vee P_3) \Leftrightarrow \neg P_1 \wedge \neg P_2$

13.9 $\neg (\neg P) \Leftrightarrow P$

14- Elimination of redundancy

14.1 $P \wedge P \Leftrightarrow P$

14.2 $P \vee P \Leftrightarrow P$

14.3 $P \wedge true \Leftrightarrow P$

14.4 $P \vee false \Leftrightarrow P$

14.5 $P \wedge false \Leftrightarrow false$

14.6 $P \vee true \Leftrightarrow true$

14.7 $P \wedge \neg P \Leftrightarrow false$

14.8 $P \vee \neg P \Leftrightarrow true$

14.9 $P_1 \wedge (P_1 \vee P_2) \Leftrightarrow P_1$

14.10 $P_1 \vee (P_1 \wedge P_2) \Leftrightarrow P_1$

Examples to use this transformation rules can be found in Elmasri [2000]. Relational algebra optimization algorithms can be found in smith [1975].

All these rules can be used heuristically to transform a query into another simpler form. Our simpler target form must be in compliance with the following heuristic rules:

- Perform joins, which lead to a smaller size of intermediate result first.
- Perform selection and projection as early as possible to minimize the processing time.
- Avoid Cartesian product operations
- Identify sub-trees whose operations can be pipelined.

In order to accomplish these heuristic deductions a heuristic engine have to be used as illustrated in Figure 2.1.

2.4.2 Query trees

This is based on the representation of the query as a graph, called a query graph or connection graph [Ullman, 1982]. SQL queries are transformed into relational algebra query that can be further transformed into a query

tree. Tremendous number of query trees can represent some complicated queries. In order to minimize the space that have to be explored, the optimizer have to consider many constraints such like:

Selection and projection constraint:

Selection and projection within a query tree have to be moved all the way down the tree as much as possible. In other words, selection and projection have to be done before the joining. This will save the unnecessary processing time for the unrelated tuples and unrelated attributes too. A restructuring algorithm to implement that was written by Ullman [1982].

The following database schema will be used for the few next examples:

Employee (Ename, Eage, Esal, Edno)

Department (Dno. Dname, Dfloor, Dano)

Account (Ano, Atype, Abalance, Abno)

Bank (Bno, Bname, Baddress).

As an example consider the following query:

Select Ename, Dfloor

From Employee, Department

Where Employee.Edno=Department.Dno and Esal > 50,000

There are many ways to represent the query tree for this SQL query as depicted in Figure 2.4.

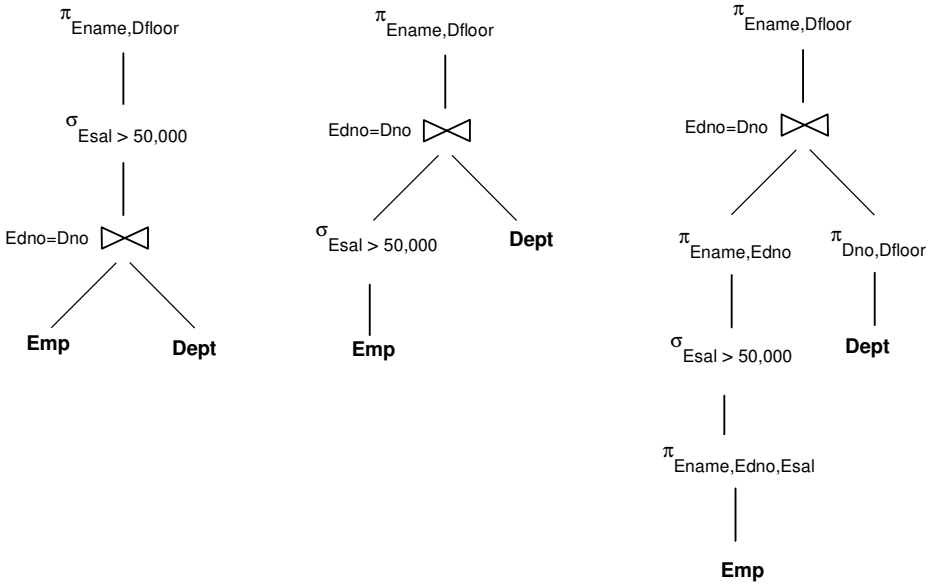


Figure 2.4. Query trees

As can be seen from Figure 2.4 that the most efficient method or plan is the one in the right-hand side where projection and selection is performed first.

Assume there is a multiple relations that are needed to be used in multiple joins in a query. Then we may benefit from some algebraic properties of join to determine certain facts. As an example, Commutativity ($R_1 \bowtie R_2 = R_2 \bowtie R_1$) can be used to determine which relation should be inner relation and which relation should be outer relation in the join.

As a second example, Associativity ($(R_1 \bowtie R_2) \bowtie R_3 = R_1 \bowtie (R_2 \bowtie R_3)$) can be used to determine the order at which joins must be executed, usually the most inner joins will be executed first.

The alternative trees that might be formed using the associativity and commutativity properties are enormous. The following constraint helps to minimize them.

Cross product should not be used:

Cross product is never to be used unless it is stated explicitly to do so. Relations within a query are concatenated through a join. As an example, consider the following query.

Select Emane, Dfloor, Abalance

From Employee, Department, Account

Where Employee.Edno=Department.Dno and

Department.Daco=Acount.Ano.

Three different query trees as shown in Figure 2.5 can form this query.

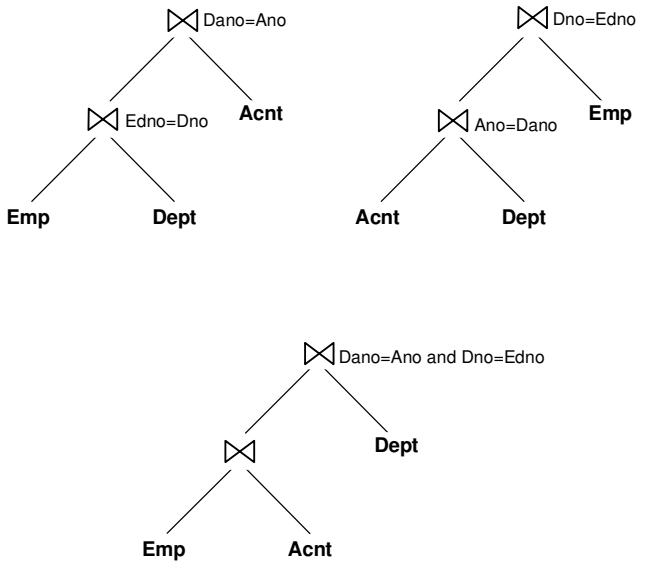


Figure 2.5. Join query trees

Notice that the last tree posses a cross product. Therefore, the space search module will ignore it. This example illustrates the use of associativity property. Cross product join is always not considered because of its huge size result.

The inner operand in a join have to a relation:

Inner operand of a join must be a relation. We must prune trees with intermediate result as inner operand. It has been proved by simulation that using relations as inner operand in a join increases the use of indexes and at the same time intermediate results that are located as outer operand will facilitate execution of the joins in a pipelined manner.

As an example consider the following example:

Select Ename, Dfloor, Abalance, Baddress

From Employee, Department, Account, Bank

Where Employee.Edno=Department.Dno and

Department.Dano=Account.Ano and Account.Abno=Bank.Bno

Figure 2.6 shows the three possible queries arrangement to execute this query.

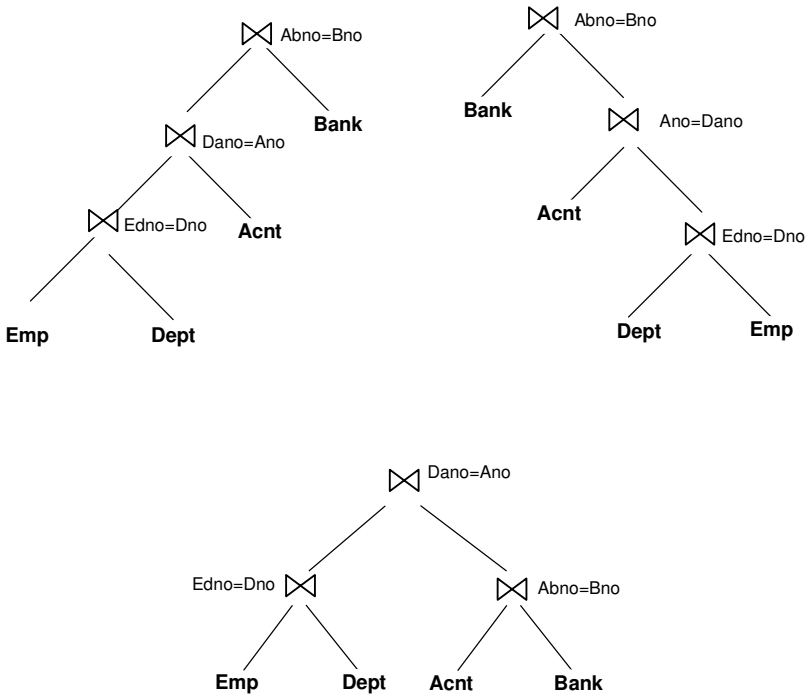


Figure 2.6. Left deep, right deep, and bushy trees

Note that the first tree in Figure 2.6 satisfy this constraint. The first tree is called left deep tree of multiple joins. The second tree is called right deep tree. And the third is called bushy tree. Discussion of these trees can be found in Ioannidis [1991].

2.5 Method Structure

Method structure is necessary to be applied when considering the type of join we would like to perform. There are many types of joins. But the

most famous three types of joins are nested loop join, merge join, and hash join. One of them is more optimal than the others within its context of use. For abbreviation purposes two methods only will be discussed, they are nested loops and merge join:

2.5.1 Nested loops

When we want to join two relations, the relation that we have its tuples as a basement for comparison is called external relation and it's tuples are read first. The relations from which tuples are compared to external relation tuple's are called internal relation, and its tuples are read secondly.

In nested loops we compare every single tuple in the external relation with every single tuple in the internal relation. We first chose a tuple from the external relation then compare it with all the tuples from the internal relation one by one. If the internal relation is sorted in the joining key and the joining operator is of equality type with no repetition, then the complexity will be n1+n2. If the internal relation is not sorted, then the complexity will reach up to n1*n2 in the worst case. Where "n" represent number of blocks.

2.5.2 Merge Join

If the external relation along with the internal relation are both sorted in the joining keys, then we may perform the join most efficiently by scanning the external relation only once with also scanning the internal relation only once. This method of join is recommended when both joined relations are sorted as a primary key in the joining key.

There has been suggested other types of joining such like the hash partitioning join. In this joining we partition the tuples of each of the relations into many sets that have the same corresponding hash value with regard to the join attributes. But the most popular types of joins are nested loops and merge joins.

Which to choose, depends in many factors such like. Do we have indexes for the joining key? If we do, is it for both relations or just only for one of them? In case that we do not have indexes for neither one of the joined relations, then we have to find the cost of sorting. Two options for sorting, either sort one relation or sort two relations. Implement internal or external sorting [Knuth, 1973]. And for each option, find the cheapest cost with regard to nesting loops join or merge join to decide what the most cheapest combination will be? The optimizer should be able to

decide the join method. Which depends on the presence of indexes, type of join if it is of equality or in-equality type, and the size of the relations.

2.6 Size & Selectivity Estimators

The selectivity of an attribute from a relation that participates in a join is necessary to estimate the size of the intermediate result and the size of the final result. Real life application may depend in frequency of more than one individual attribute. But all existing DBMSs that exist now do not support multiple attribute frequency that may be needed for a join due to the fact that it is expensive to have them for a single relation. Most commercial systems depend on a technique called histogram, which we will discuss. Histograms contain the database statistics about the frequency of the database attributes. There is a trade-off between the accuracy of the statistics and the cost of maintaining it [Piatetsky, 1984]. Histogram techniques classify a number of frequency values in buckets. For an attribute A in a relation R, the domain of A is grouped into buckets. Each bucket B in the histogram will have many different values frequencies. That is, for a value $V_i \in B_i$, the frequency f_i of V_i is denoted by $\sum V_j \in B \ f_j / |B|$.

Some DBMSs assume uniform distribution assumption over all the values in a relation. These types of histogram are called trivial [Selinger, 1979].

The error percentage in estimation in this type of histogram is very high [Christodoulakis, 1984].

To reduce the error margins, we group the frequency of attributes for a domain in several numbers of buckets. Each bucket estimation is done separately. In this way the error in the estimation is reduced. As an example, suppose we have a relation as depicted in Table 2.1.

Table 2.1. Employee relation

Emp_No	Name	Dept	Salary
113	Ahmad	Education	JD 6000
125	Hassan	Health	JD 4500
548	Ali	Admin	JD 6500
284	Omar	ISO	JD 7000
246	Khalid	Health	JD 3000
684	Sami	Education	JD 4000
964	Hamad	ISO	JD 5500
574	Emad	Supply	JD 6000
347	Majed	Finance	JD 7000
246	Hiam	Education	JD 8500
475	Jasem	ISO	JD 4700
254	Nadiah	Supply	JD 6500

Frequency of department in Employee relation is illustrated in Table 2.2.

Table 2.2. Department frequency in employee

Department	Frequency
Admin	1
Education	3
Finance	1
Health	2
ISO	3
Supply	2

Now the frequency is grouped into consecutive buckets as shown in Table 2.3, which called equi-width histograms [Kooi, 1980]. This type of histograms is the most popular type that is used in most commercial DBMSs.

Table 2.3. Equi-width histogram

Buckets	Department	Frequency in Bucket	Approx. Freq.
Bucket 1	Admin	1	1.66
	Education	3	1.66
	Finance	1	1.66
Bucket 2	Health	2	2.33
	ISO	3	2.33
	Supply	2	2.33

The margin of error could be further reduced. You may notice that range in equi-width histogram is taken serially. Equal numbers of entries are included in each bucket, independent of the frequency number of these values. If we take in consideration the number of frequency for each entry in the buckets, and form buckets based in those (numbers of frequency) where each bucket have to contain a specific equal number of total frequencies, then the estimation will be more accurate as illustrated in Table 2.4 . This type is called Equi-depth histogram [Piatetsky, 1984]. Note that frequencies are grouped into non-consecutive range of

frequencies. This type of histogram adds more complexity to the generating algorithm. Hence, it is not used in commercial DBMSs.

Table 2.4. Equi-depth histogram

Buckets	Department	Frequency in Bucket	Approx. Freq.
Bucket 1	Admin	1	1.5
Bucket 2	Education	3	3
Bucket 1	Finance	1	1.5
Bucket 1	Health	2	1.5
Bucket 2	ISO	3	3
Bucket 1	Supply	2	1.5

More advanced histogram approaches have been suggested such like serial histogram [Ioannidis, 1993]. In this histogram all the buckets frequencies have to be ordered and maintained ordered serially throughout the life of the related relation. That is all frequencies in one bucket have to be greater than or less than all other frequencies in different buckets. Since the frequency of an attribute is independent of it's value, it is hard to maintain them grouped in order all the time.

To improve this histogram, an extension of serial histogram has been suggested, it is called End-biased histogram. This histogram uses different buckets for the highest and lowest values of frequencies and keeps the mean frequencies in a separate bucket. In this way all frequencies are kept serially and the complexity that exist in the previous histogram is eliminated. This histogram is simple and preferred over serial histogram.

Other types of histograms like piecewise linear histograms are discussed by Yu [1994].

The size of the intermediate relation in a join is calculated using size estimator, if necessary, by using the information obtained from the selectivity estimator [Lipton *et al*., 1990].

2.7 The Cost Model

During execution of a query many factors have to be considered in estimating the cost. Some of these factors have been discussed thoroughly through out this dissertation. These factors are as follows:

- Communication cost if necessary: this cost is the dominant factor in distributed systems, WAN, and MAN.[Valduriez, 1984]
- Memory cost: The number of memory buffers needed during the query processing.
- Intermediate files storage cost.
- Computation cost: which includes sorting, merging, and searching.
- Secondary storage media access cost: The cost of accessing secondary storage such like hard disks or tapes. This cost depend in the type of access mechanism such like: accessing through index or not. Primary of secondary indexes. Contiguous memory location or random … etc.

2.8 The Planner

The planner duties is to identify the possible promising solution plans explored by algebraic space and method structure modules, then pass them to the cost module to find which one is the cheapest.

The cost module calculates the cost of each received plan based on data obtained from size estimator and selectivity estimator modules.

There are different strategies that a planner could employ. The most popular strategy is the dynamic strategy, which is used by most commercial DBMSs. It will be discussed later in this section.

Other strategies are based on random algorithms like iterative improvement [Swami, 1989], simulated annealing [Ioannidis, 1987] and two phase optimization (2PO) algorithm [Ioannidis, 1990]. They operate by searching a graph where nodes represent alternative paths and plans that can be used to answer the query. Each node is associated with a predefined cost. The goal of these algorithms is to try to find the cheapest path (Plan) with minimum time. The returned path (plan) is not guaranteed to be the cheapest, but it is efficient in dealing with queries that have huge numbers of joins (ten or more joins). For small number of joins it is preferred to use strategies that are based on dynamic algorithms, as they give more efficient cheapest plan to solve a given query. Random algorithms are not of our concerned here at this dissertation.

Lately, some researchers have tried to explore the possibilities of applying some heuristic algorithms such like A* algorithms to find the cheapest query plan solution [Yoo, 1989].

In what follows we will discuss the most popular strategy for the planner in the optimizer, which is based in dynamic algorithm.

The dynamic algorithm R was first written by Selinger *et al*. [1979] and since then it was used in commercial systems in different appearances. This algorithm builds all the promising query trees, pruning all trees that are not found to provide an optimal plan.

Algorithm 2.1 illustrates the steps that are implemented in "R" algorithm. Actions implemented by R algorithm are grouped as per the following steps:

Step 1. All the possible access to all relations in the query are identified. Their cost is obtained from the cost module. All this information is retained in a table to be used in step 2.

Step 2. For each join in the query, all possible ways to perform the join between two relations, with the help of the relations access methods stated in step 1, are examined and it's cost is calculated.

Algorithm 2.1. R Algorithm

Input : QT : Query tree with n relations
Output : The result of algorithm execution
Begin
  For each relation $R_i \in QT$ do
  Begin
    For each access path AP_{ij} to R_i do
      Determine cost(AP_{ij})
    End-for
    Best_$AP_i \leftarrow AP_{ij}$ with minimum cost
  End-for
  For each order (R_{i1}, R_{i2}, ... , R_{in}) with i =1, ... , n! Do
  Begin
    Build strategy (...((best $AP_{i1} \bowtie R_{i2}) \bowtie R_{i3}) \bowtie$ ... $\bowtie R_{in}$)
    Compute the cost of the strategy
  End-for
  Output $\leftarrow$ strategy with minimum cost
End {R Algorithm}

Step 3. The paths and the promising query trees are evaluated using the costs obtained in step 2 to identify the cost of each plan with different combination.

Step 4. The cheapest or best plan from step 3 is chosen for execution. There is one important factor within the implementation of this algorithm called interesting or attractive order. The sorting order of an intermediate relation generated by an intermediate join might be in the desired order in which subsequent join might be based on [Ioannidis, 1990]. One should keep in mind, if this can be fulfilled, it will save the sorting time for merge-join or nested loop. The path through which this attractive order can be achieved may require more cost than other paths, but it may save more subsequent cost in sorting time than choosing a cheaper path that may require more time for sorting.

The algorithm' first loop evaluates the cheapest cost to each relation separately.

The second loop will test the paths of *n!* Permutation for *n* relations. The test start with the most inner joins in the loop. It is implemented by testing each relation join with each other relation before adding the third, then the forth, and so on. Until all permutations are tested. The algorithm prunes paths with cross product, and also prunes paths that are commutatively equivalent but with higher cost. Therefore, the total tested paths will end up to 2^n rather than *n!*.

3. METHODOLOGY

3.1 TEMPORAL RELATIONAL DATABASE SCHEMA & MODELS

3.1.1 Overview

Temporal enabled file structure of temporal relational database provides a solid concrete foundation for query processing and optimization.

We will discuss possible efficient file structures for different types of temporal relational databases. This discussion will be general.

Meanwhile, we will concentrate in our recommended file structure that works efficiently for our indexing structure and joining algorithms. Also we will discuss temporal file structure. And finally, available temporal relational database schemes will be discussed along with our suggested enhancement to the existing schemes in which it will accommodate the temporal nature adequately. Temporal models and schemes have been discussed by Segev [1988], Delaney [1992], Elmasri [1993], and Gadia [1988A].

3.1.2 Temporal Relational Database File Structure

We have to use fixed length not spanning records for the database file structure design. This will enable us to use indexing and sorting without moving the data from its original memory location in case of entering the ending time stamp (for both time stamps, valid and transaction timestamps). Using fixed length not spanning records will eliminate extra processing time that may be required using other dynamic length spanning records. Also it will facilitate an easy file re-organization and garbage collection during the regular temporal database management information system maintenance.

Time stamps in regular traditional databases are of fixed size and format (dd-mm-yyyy, hh-mm-ss). While real life applications require different granularity for different application. As an example, consider a personnel module in an application for a firm. Hiring new employee would require time stamp of "day" granularity. If we include hours, minutes, and seconds, we would cause a waste of memory space, in addition to unnecessary extra processing time during query run-time. Also, there are application that are based on fiscal year (e.g. budget, planning ...etc). One could imagine the inconvenience of using more than "Year" granularity in a firm computerized applications. Especially if that firm have many branches with many departments.

Temporal aspect is applied only to all fields that are subject to change in status. In order to record the history for a specific attribute in the database, all actions done to this attribute have to be recorded. This will provide us with a powerful analysis tool from which we can predict trends and guide our decision. Since temporal database applications have to have time stamps accompanied with all activities, it would be desirable to minimize time stamps as much as possible.

Temporal database management systems need to facilitate built-in data constructs that represent all possible granularities with minimum possible usage of bits.Table 3.1 shows the suggested time granularity and its required number of bytes for representation.

Table 3.1. Needed bytes for different granularities

Granularity	**Included Granularities (By Default)**	**Time Format**	**Needed Bytes**	**Possible Range From – To**
Year	None	yyyy	1	2000-2255
Year (Ext.)	None	yyyy	2	0000-9999
Month	Year	yyyy/mm	2	2000/01-2255/12
Month (Ext.)	Year	yyyy/mm	3	0000/01-9999/12
Day	Year-Month	yyyy/mm/dd	3	2000/01/01-2255/12/31
Day (Ext.)	Year-Month	yyyy/mm/dd	4	0000/01/01-9999/12/31
Hour	Year-Month-Day	yyyy/mm/dd /hh	4	2000/01/01/00-2255/12/31/23
Hour (Ext.)	Year-Month-Day	yyyy/mm/dd /hh	5	0000/01/01/00-9999/12/31/23
Minute	Year-Month-Day-Hour	yyyy/mm/dd /hh/nn	5	0000/01/01/00/00-9999/12/31/23/59
Second	Year-Month-Day-Hour-Minute	yyyy/mm/dd /hh/nn/ss	6	0000/01/01/00/00/00-9999/12/31/23/59/59

As can be seen form the table, if the required granularity is a day for a certain application, then year and month have to be included to guarantee the uniqueness of the day. If we do not include year and month, then they may be repeated with previous or next months in previous or next years. The same rule applies for all granularities. If we abide by the above table when designing a temporal application, we will improve usage of memory by reducing it. Consequently, optimize the processing time.

Table 3.2 lists the improvement in the required memory in comparison with the traditional Relational Database Management System software, Microsoft Access. Improvements are determined by calculating the decrease percentage in size. Microsoft Access uses fixed size 8 bytes to represent the date as specified per Microsoft technical instruction in Microsoft Access 97.

Table 3.2. Memory comparison between Access and the suggest model.

Granularity	**Access Size (In Bytes)**	**Suggested New Size (In Bytes)**	**Approximate Improvements**
Year	8	1	88 %
Year (Ext.)	8	2	75 %
Month	8	2	75 %
Month (Ext.)	8	3	63 %
Day	8	3	63 %
Day (Ext.)	8	4	50 %
Hour	8	4	50 %
Hour (Ext.)	8	5	38 %
Minute	8	5	38 %
Seconds	8	6	25 %

Three facts can be extracted from the above tables (Table 3.1 and Table 3.2). First, granularity can be set according to the application needs. Second, save in memory ranges between 25 % - 88 % are achieved by using the suggested model. Third, Saving in the size of time stamps will lead to less processing time. Therefore, it will improve the performance of the queries. Discussion of time data was carried out by Skjellaug [1997] Consideration of optimizing the usage of memory has to be considered in temporal relational database since time stamps will be used heavily in temporal databases.

Represented time fields have to be reduced to save space since temporal databases are usually huge in their size. Great amount of memory would be wasted if we use one defined time stamp regardless of the needed granularity and that will affect the performance negatively, in addition to wasted memory. Flexible size capability for the size of the temporal time stamps according to necessity would save memory and improve performance of the queries.

3.1.3 Temporal Relational Database Schemes

Many literatures have suggested different types of relational database schemes. Steiner [1998] have discussed many data models. Within his temporal relation data models, he has discussed schemes that are based on

tuple time stamping, and attribute time stamping. Detailed evaluation for different temporal databases are discussed by Ahn [1986]. Experiments on the performance of different types of temporal databases were carried out by Goralwalla [1995]. Similar work has been done in this dissertation, but we added our test result with regards to memory usage.

Basically, there are two main approaches for modeling temporal relational databases [Goralwalla, 1995] and [Ahn, 1986]. They are as follows:

1- Attribute time stamp: the time is attached to attribute values of a relation and the histories of an attribute are included in a set of triplet-valued. Example is shown in Table 3.3 . In this example a triplet is of the form <[*l, u*), *v*> where *l* represent lower time bound, *u* represent upper time bound, and *v* represent the value of the attribute.

Table 3.3. Employee relation

E#	Ename	Department	Salary
111	Hamd	{<[1984/01,1985/01), Shoe>; <[1985/01,now], Toys>}	{<[1984/01,1985/06), 20K>; <[1985/06,now], 25K>}
222	Ayah	{<[1986/01,now], Sales>}	{<[1986/01,now), 30K>}
333	Sami	{<[1989/01,now], Toys>}	{<[1989/01,1990/06), 32K>; <[1990/06,now], 40K>}

2- Tuple time stamp: It is divided into two subcategories:

(A)- where a single relation is used to hold all its pertaining time varying attributes along with non-temporal attributes. Time stamps are represented in “From” and “To” fields. The time stamps are used to

indicate changes in any temporal attribute within a tuple in a relation. Table 3.4 depicts this type.

Table 3.4. Tuple time stamping

SSNo	Ename	DoB	Salary	Dept	From	To
111	Muna	1973/01/14	6,000	D1	2000/01/01	2000/05/31
111	Muna	1973/01/14	6,000	D2	2000/06/01	2000/12/31
111	Muna	1973/01/14	7,000	D2	2001/01/01	2001/06/30
222	Hani	1970/03/07	5,000	D1	2001/04/01	Now

B - Time varying attributes are distributed over multiple relations, and non-temporal attribute are gathered in separate relation. To illustrate that, we used the same example in Table 3.4. The relation is reorganized to fit this scheme as in Tables 3.5.

Table 3.5a. Non-temporal attribute

SSNo	Ename	DoB
111	Muna	1973/01/14
222	Hani	1970/03/07

Table 3.5b. Salary temporal attribute

SSNo	Salary	From	To
111	6,000	2000/01/01	2000/12/31
111	7,000	2001/01/01	2001/06/30
222	5,000	2001/04/01	Now

Table 3.5c. Department Temporal attribute

SSNo	Dept	From	To
111	D1	2000/01/01	2000/05/31
111	D2	2000/06/01	2001/06/30
222	D1	2001/04/01	UC

First approach uses Non-First Normal Form (N1NF) where second approach uses First Normal Form (1NF). Many researchers like Gadia [1988B] support first approach. Second approach is supported and modeled by Snodgrass [1987].

Attribute time stamping needs more work for incorporating indexing for query optimization and other works related to updating. Great amount of overhead is expend for operating functions like Pack, Unpack, Triplet-formation, and Triplet-decomposition. For more details refer to Goralwalla [1995].

Also, Attribute time stamping violate the first normal form where the only attribute values permitted by 1NF are single atomic or indivisible values. For these reasons, Attribute time stamping is excluded from our consideration for the indexing and joining algorithms. That's will leave us with option 2 (tuple time stamping). In what follows we will compare between the two possible types of tuple time stamping. And that is:

1st- A single relation is used to hold all its pertaining time varying attributes along with non-temporal attributes. Time stamps are

represented in “From” and “To” fields. The time stamps are used to indicate changes in any temporal attribute within the relation. We will refer to this type as (TTSR) abbreviation for Tuple Timestamp Single Relation.

2nd- Time varying attributes are distributed over multiple relations, and non-temporal attributes are gathered in separate relation. We will refer to this type as (TTMR) Tuple Timestamps Multiple Relations.

Fortunately, results with regards to performance are already available in [Goralwalla, 1995]. Different queries that may cover most combination of possible requirement were tested against TTMR and TTSR. The tests cover current status data and historical data. It has been found that TTMR surpass TTSR order of magnitudes in performance for both current status and historical data, with regard to execution time.

But comparison with regards to used memory is not available and will be carried out in this dissertation.

If we assume that a database D consists of a set of relation

$D=\{T_1, T_2, T_3, \dots, T_n\}$

Each relation consists of different fields, these fields can be grouped into 4 main groups, they are Key field/s, Non-Temporal fields (Unchangeable), Temporal fields, and Timestamp. They are represented

by *K*, *U*, *M*, and *L* respectively. Key fields are always non-temporal fields. Each group may consist of at least one field or more fields. Hence:

$K = \{k_1, k_2, k_3, \ldots, k_n\}$

$U = \{u_1, u_2, u_3, \ldots, u_n\}$

$M = \{m_1, m_2, m_3, \ldots, m_n\}$

$L = [l_1, l_2]$, where l_1 is the lower bond (start time) and l_2 is the upper bond (end time).

Therefore: A relation consists of a set of subsets as follows:

$T=\{K, U, M, L\}$

Let S_i : Size of field *i* in bytes where *i* may be *K*, *U*, *M, or L* field

$C(t)$: The cost of a tuple in total number of bytes, *t* may be a tuple of TTMR type, or TTSR type.

P : Probability for a temporal attribute to be updated (changed) measured in number of times it may be updated within a specific period of time.

The cost of representing the temporal relation in TTSR form will be:

$$C\,(\text{TTSR}) = \sum_{i=1}^{X} S_i \; * \; \sum_{i=1}^{Y} P_i \qquad \ldots\ldots\ldots \quad (1)$$

Where x represent number of all fields in a tuple and y represent temporal fields only. Suppose that $y_1 \in M$ and $y_2 \in M$, then $\exists \neg C(y_1) \Rightarrow C(y_2)$, that

is, we assume that temporal attributes are independent and change in one temporal attribute do not imply change in another.

The cost of representing the temporal relation in TTMR form will be:

$$C\,(\text{TTMR}) = \sum_{i=1}^{Y} (S_K + S_i + S_L) * P_i \qquad \ldots\ldots\ldots \quad (2)$$

A separate relation exists for non-temporal attribute' tuples. Since there exist only one tuple for every key field, the size of this relation is considered to be negligible in comparison to temporal relations where we have indefinite numbers of tuples for every key field.

The total save in memory (space) for a certain period of time, say $L = [l_1, l_2]$, can be calculated as follows:

$$C\,(\text{Improvements}) = \frac{C\,(\text{TTSR}) - C\,(\text{TTMR})}{C\,(\text{TTSR})} \qquad \ldots\ldots\ldots \quad (3)$$

With respect to assumptions in formula 1 and formula 2 above. It can be seen from formulas 1, 2, and 3 that the save in memory would be directly proportional to P_i and to the number of attributes in M set.

Testing:

The above formulas (formulas 1, 2, and 3) have been used to carry a comparison test between TTSR approach and TTMR approach using the

same schemes that have been used in execution performance test by Goralwalla [995]. These schemes are as follows:

TTSR:

Emp (ssno, ename, address, salary, skills, dname, from, to)

Dept (dno, dname, budget, manager, from, to)

Proj (pno, pname, budget, from, to)

TTMR:

Emp U (ssno, ename, address)

Emp M1 (ssno, salary, from, to)

Emp M2 (ssno, skills, from, to)

Emp M3 (ssno, dname, from, to)

Dept U (dno, dname)

Dept M1(dno, budget, from, to)

Dept M2(dno, manager, from, to)

Proj U (pno, pname)

Proj M1(pno, budget, from, to)

The sizes in bytes for each field are as follows:

Attribute	**Size in bytes**

Ssno	2
Ename	4
Address	4
Salary	6
Skills	7
Dname	7
From	2
To	2
Dno	2
Dname	3
Budget	6
Manager	7
Pno	2
Pname	3

The probability of temporal attribute changes is measured by the frequency of changes per year. For simplicity it is assumed to be one change for all the temporal attributes per year independently. Calculating the cost of storage using formulas 1-3 above according to the above assumptions have been carried out. The result of the calculation shows that save in storage cost ranges from 20%-71% in favor of TTMR approach as illustrated in Table 3.6.

Table 3.6. Improvement of TTMR schema over TTSR schema.

Relation Name	**No. of Temporal Attributes**	**Memory Save %**
Emp	3	71 %
Dept	2	43 %
Proj	1	20 %

Also, It has been proved that saving in space may exceed 100% with very higher P_i.

The tests for memory usage were chosen to be compatible with the testing environment of the test done with regard to the execution time performance [Goralwalla, 1995]. Most probably other tests for other applications will give different results. Also, changing P_i will affect the result. In fact, the real test will be the real life applications.

In regular snapshot (transaction) database, updating are done easily because there is only one key field that is never changed. Therefore; regular daily transactions (Add, Delete, and Update) are implemented in a straightforward manner.

Unfortunately, this is not the case in temporal databases. Assume a non-temporal relation in non-temporal database that has only one key field. In order to implement the same relation as a temporal relation in the corresponding temporal relational database, then we have to add extra 3 key fields in addition to the main key field and that will total up to 4 key fields. These key fields are as follows:

- Main Key field, also called Surrogate in temporal database,
- Time Attribute field,
- Starting Time,

- and Ending Time.

To deal with all of these four keys in regular daily transactions (Add, Delete, and Update) and keeping the existing sorting intact for efficient optimized query processing is a challenging task. This task is much easier for transaction (roll back) databases since we anticipate the transaction start timestamp to take a rhythmic pattern. But it would be much more difficult for valid time (historical) databases were neither start timestamp, nor ending timestamp are predictable. Therefore; indexing and file structure for valid timestamps are forced to be different than those that are used for transaction timestamps.

In this dissertation we are concerned with relational roll back databases. In which we support through indexing methodology explained in section 3.2. And also supported in special algorithms for joining in section 3.3. They all come together as one strongly related subject in optimization of temporal relational queries.

Therefore, in this section we will elaborate only on regular daily transactions that are related to roll back databases that are based on transaction timestamps.

In Figure 3.1 we illustrate the suggested processing of regular daily transactions with regard to temporal databases that are used for roll back databases.

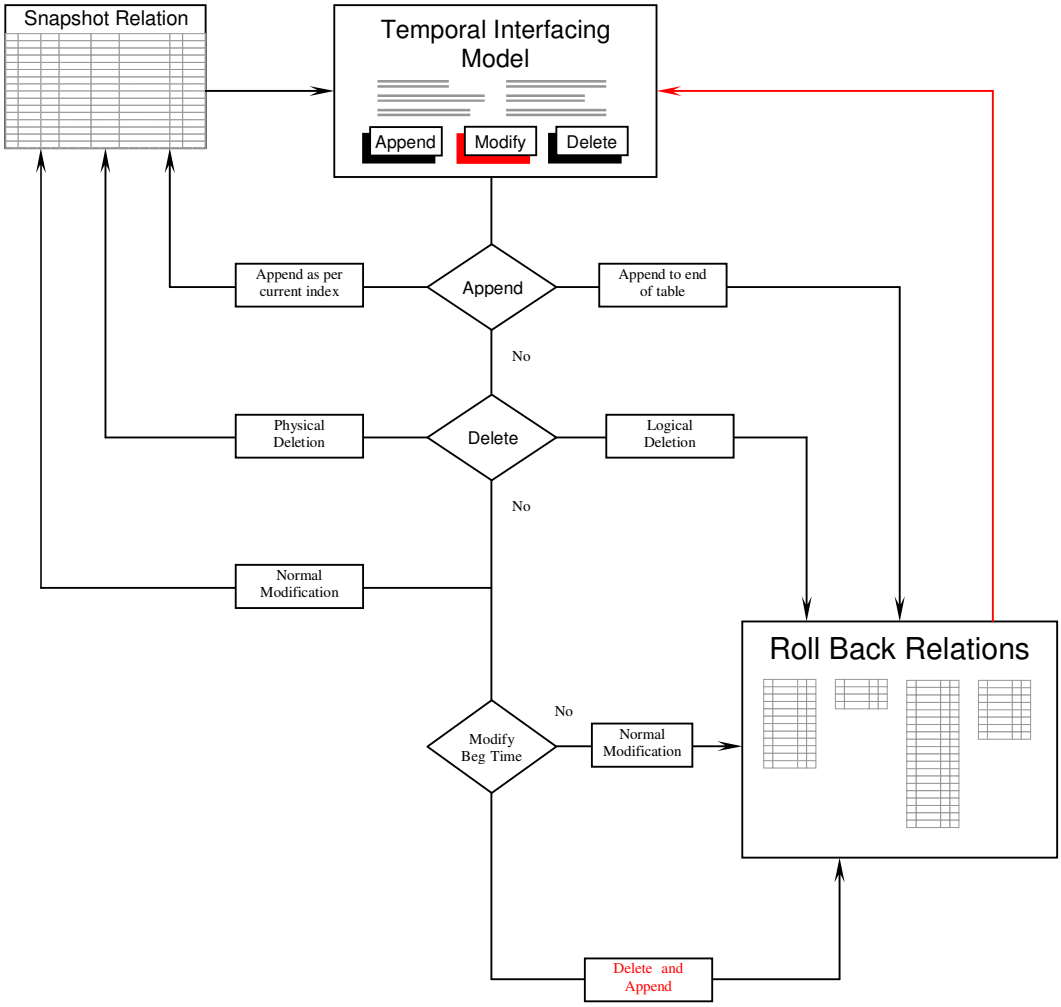


Figure 3.1. Daily transactions in roll back database

We use quasi-append only approach for roll back relations.

We explain how this model works through the following example:

Example: For simplicity, assume that current time is 2001-02-01, the company starts operation in 1997-01-01, and changes in salary

or departments happens in the beginning or the middle of the year as illustrated in Table 3.7.

Table 3.7. Employee snapshot relation

SSN	Name	DOB	Hire-Date	Salary	Sal-TS	Dept	Dep-TS
111	Ahmad	1960-05-27	1997-01-01	JD 6000	2000-02	D1	1998-08
222	Mona	1968-11-04	1997-01-01	JD 7200	2000-02	D3	1997-01
333	Khalid	1958-07-13	1999-01-01	JD 5520	1999-07	D2	1999-07

The following abbreviations are applicable for the temporal relations:

TS : Transaction Time Start

TE : Transaction Time End

S : Surrogate (Key field in temporal relation)

A : Temporal Attribute

Table 3.8. Employee-salary temporal relation

TS	*TE*	*S*	*A*
1997-01	1997-07	111	JD 4000
1997-01	1998-01	222	JD 4500
1997-08	1998-01	111	JD 4500
1998-02	1999-01	111	JD 5000
1998-02	1999-01	222	JD 5000
1999-02	2000-01	111	JD 5500
1999-02	2000-01	222	JD 6000
1999-07	UC	333	JD 5520
2000-02	UC	111	JD 6000

2000-02	UC	222	JD 7200

Table 3.9. Employee-department temporal relation

TS	***TE***	***S***	***A***
1997-01	1998-01	111	D2
1997-01	UC	222	D3
1998-02	1998-07	111	D3
1998-08	UC	111	D1
1999-07	UC	333	D2

Append transactions:

As illustrated in Figure 3.1. The append transactions are added to snapshot relation with respect to existing index key, usually its correspond to surrogate key in temporal relation. Additions of records to corresponding temporal relations in roll back relations are done through adding the new records to a buffer table called Temporal Buffer Table (TBT). This table are vital in temporal databases for two reasons. First, It is a must in multi-user environment to synchronize work and append them from that buffer to the related relation one by one and allow to synchronize locking and unlocking to the related temporal relation. Second, this buffer table is closed at the end of the specified one time granularity (chronon). At the same time another TBT is opened for the

next chronon. Before we transfer the appended data from the closed TBT into the related temporal relation, we can sort the records in a second key, third key, or forth. Therefore; create a primary index with multiple keys that helps to accommodate different join process, as we will see in section 3.3. This will help in a query optimization.

Delete transactions:

Deletion of records in snapshot relation can be implemented physically. The related records in the roll back relations will be deleted logically by marking out the deleted records as deleted. These logically deleted records can be permanently deleted during the maintenance period for temporal database.

Modifying transaction (updating):

Since this scheme is build on quasi-append only database and time stamps are based on transaction time, then updating a temporal attribute with a new value will be dealt with in two steps. The first step is to locate the corresponding record in roll back relations and close it by entering the (current time – one chronon) as ending timestamp. The second step will be to append a new record to the end of quasi-append only temporal relation. This record will have a Starting timestamp and "UC" ending

timestamp. Also modify the related TS field in snapshot relation accordingly.

Main key along with related TS value can be used to locate a record in the first step. As an example, suppose we have reached the date of June 2001 and in the snapshot relation "Mona" has a raise in salary from the existing JD 7200 into JD 8000. We can see from the relation in Table 3.7 that "Sal-TS" value is 2000-02 and her "SSN" is 222. Hence we go to Employee-salary temporal relation in Table 3.8 and locate "TS" with value of 2000-02 and "S" value equal to 222. We change "TE" value into 2001-05 and append a new record through TBT with the following values 2000-06, UC, 222, and JD 8000 for "TS", "TE", "S", and "A" values respectively. We also updated "Salary" and "Sal-TS" values in Employee snapshot relation from the old values into a new values of JD 8000 and 2001-06 respectively.

Updating the "TS" value in temporal relation have to be done with extreme caution. It is implemented by marking up the modified record with deleted and adding up the corrected record into a dedicated overflow table in order to keep the primary key index intact physically. These kinds of overflow tables can be integrated with related temporal roll back relations during the maintenance phase of the temporal database.

Temporal model does all these transactions automatically. Users do not have to worry about these details. The user will be dealing with an abstract model called Temporal Interfacing Model (TIM). This model reveals the valid current snapshot data from the snapshot relation to the user. The user may update the data and press on save or modify (update) button, TIM will know automatically what has been updated and carry out the necessary steps as explained in the updating procedures above.

TIM will react to entering new records in a similar way and will perform all necessary steps toward snapshot relation and the temporal roll back relation automatically. The XXX-TS fields in the snapshot relation can be hidden in a system table. Table 3.9 represent Employee-department temporal relation.

3.2 INDEXING

3.2.1 Introduction

Many temporal relational indexing schemes have been developed. Some of these schemes are designed to fit certain needs. Other schemes are designed generally. Nascimento [1995] used shared leaves between transaction time and valid time to save substantial space in bitemporal index. He used pointers extensively in his design in which it will add more complexity in the algorithm design.

Kouramajian [994] have designed an indexing structure called Time Index+. It is designated for data that overlap very often. In his design he used logical partitioning for buckets with regards to timestamps. This index scheme requires a huge storage space. Therefore, it is difficult to be updated.

Many other researchers have discussed the possibilities of using basically similar techniques like the technique used by Kouramajian [1994]. These

suggested techniques are Time Index, Packed R-Tree, and Parameterized R-Tree. Other suggested indexing structures are explained by Bozkaya [1998] and Spiteri [1998].

During our extensive investigation in temporal database indexing. We have not found any suggested index methodology that uses a hash function in indexing temporal relational database. This type of suggested index might be used in our model to support the joining algorithms in section 3.3. This scheme understructure was introduced in section 3.1 (Temporal Relational Schema and Models).

In append only databases only appending data is allowed, no modifications or deletion is allowed. The start timestamp for an event is entered upon the creation of tuples in transaction databases and the end timestamp is left open. When we need to indicate the change of a temporal attribute status, a new tuple with a new start timestamp is created without any change to the old tuple status. The end timestamp in the old record will be left unchanged, it will be left with open time interval. Closing time for that attribute will be calculated through finding the next corresponding record start timestamp!

The previous append-only database was modified to accommodate only one slight change. That is, closing time (end timestamp) for the old record can be modified to indicate the closing time when a new tuple for the

same temporal attribute is created with it's start timestamp equal to the old record end timestamp plus on unit of used granularity time (Chronon). This new modified scheme is called quasi-append only database [Tansel, 1993].

In this dissertation a simple temporal index structure have been designed. To implement our new suggested index we will use quasi-append only temporal relational database.

As it will be seen in this section, this design used two generic indexes approaches mixed together to form a straightforward simple efficient indexing approach for temporal relational databases.

Temporal databases main feature that distinguishes it from other databases is the fact that it depends on time more than regular classical database approaches. Temporal databases are based mainly on sequence of events. The latest event comes in after another earlier one, and so on. To create serial events in order to study its effect in connection to other events, one have to record the time at which each event took place. Timing could be represented in many ways. Most literature that have been written with regard to indexing have ignored the importance of the representation of time, and used real number to represent the sequence of events (Temporal Aspect) in their researches while representation of time have a major role in efficient temporal indexing. The suggested time

representation that we suggest in our index was previously explained in section 3.1. This representation is based the real world timing system.

3.2.2 Multi-level Clustered Index

We believe that representation of time in temporal relational database plays an important role in optimizing the query. Therefore, we have used in our approach the regular universal timing to stamp events. That is year, month, day, hour, minute, second, … etc.

Depending on the chosen granularity, as explained in section 3.1, levels will be assigned for the index. Assuming that start time "Beg Time" is used as a primary index in quasi-append only temporal relational database. ISAM (Indexed Sequential Access Method) can be adapted in multi-level clustering index. Multi-level clustered index depends in the universal timing to define levels. For example, for the sack of simplicity and without loss of generality, assume that granularity is a day level. Then a multi-level index can be designed as shown in Figure 3.2. Note that date format start with year, then month and finally day. (i.e. "yyyy-mm-dd"). In "End Time" field "UC" stands for until changed. Therefore; levels of index are organized as follows: the first level (base level) represent days, second level is dedicated for months, and the third level (top level) is dedicated for years. For an hour granularity in a relation, the first level will be hours, second level will be days, third level will be months, and

the forth level (top level) will be years. The same dividing for levels can be applied to finer granularity accordingly.

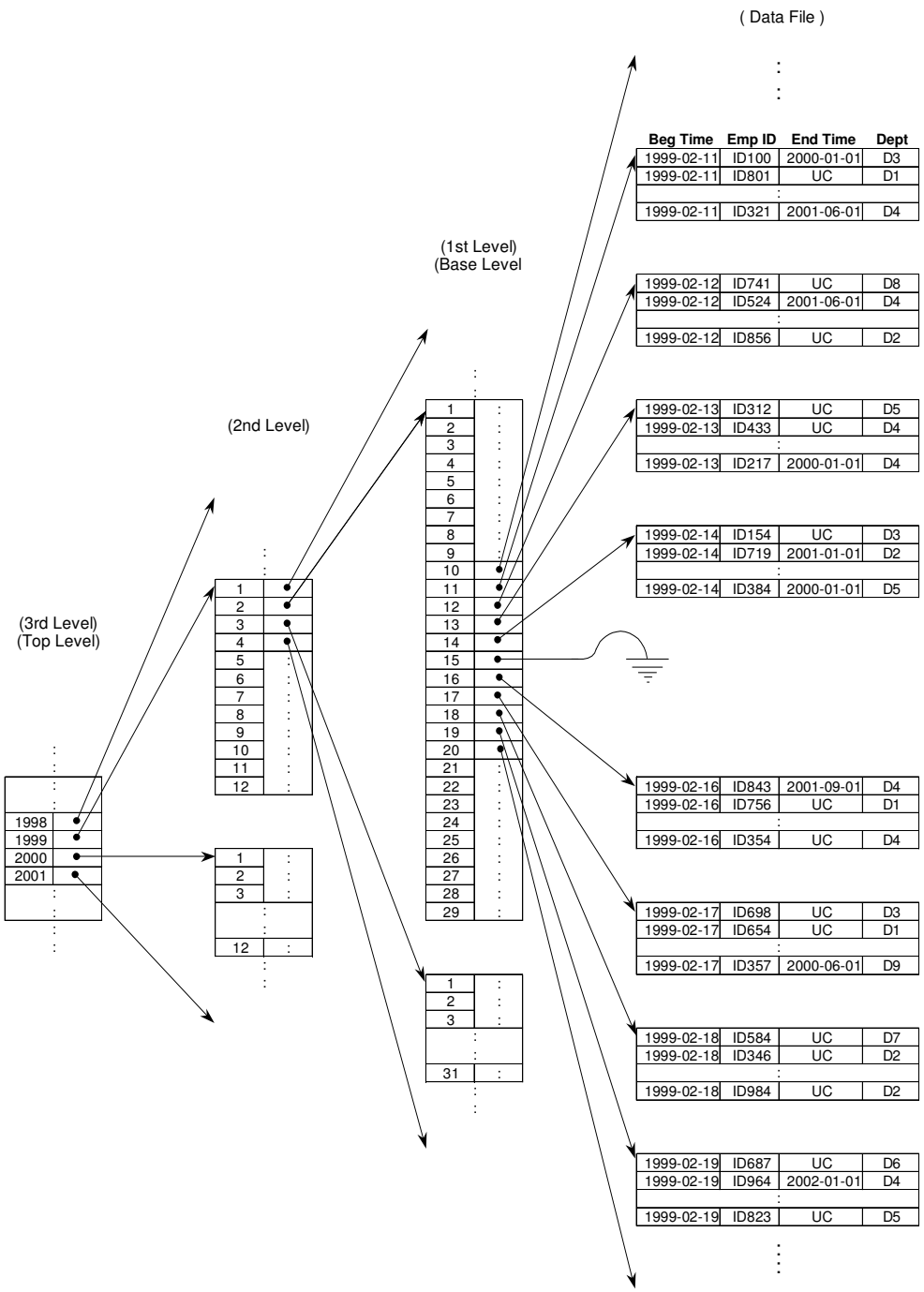


Figure 3.2. Mutli-level index

In case that no events took place at certain time period. Then that period or interval can points to Null as in date "1999-2-15" in Figure 3.2. Popescul [2000] has suggested to use clustering in identifying temporal trends in document database.

Indexes can be either dense or non-dense. In dense indexes, there exist at least one pointer that point for every key entry in data file. In non-dense indexes a range specified with two keys that have upper bound and lower bound are specified to locate a surrogate key. Dense indexes are used for small database. But to reduce the searching blocks in very large databases we used non-dense indexes. Records that are used in this index are assumed to be not spanned fixed length records.

Algorithm 3.1 explains the search procedure for a record in the data file using non-dense index based on regular dates.

Algorithm 3.1. Searching in Multi-level Temporal Index

Step 1: $D \leftarrow$ address of most outer index level (Highest Granularity)
Step 2: For i=N to 1 step −1
Go to D block ;
Search in block D for a position P of record R;
$D \leftarrow P$
Next
Step 3: Read data in block D;
Step 4: Search for record R;

At algorithm 3.1 . *R* is the record/s we are searching for. *N* is the number of levels for the multi level index.

In first step we search in the highest level index to locate the address *D* at which record/s *R* outer index is located. At step 2 multi-levels indexes are traversed all the way to the first base level in order to locate the block at which record/s *R* may be located. Steps 3 and 4 are used to locate the needed record/s within the designated block.

This Multi-level clustered index facilitates the joining algorithms efficiently. The index is maintained by default for the desired granularity plus all higher levels granularity.

In distributed systems the main factor for evaluating a query or algorithm is the communication cost. In large database (Temporal Databases) the number of block transfers from the disk is used to measure the actual cost [Ozsu, 1999] and [Silberschatz, 1997]. Therefore, in our design in this algorithm we have concentrated in minimizing numbers of disk access as much as possible.

One disadvantage in this scheme is that there are many index levels access for every date as in Figure 3.2. We need three accesses to get to the data file. If we use hour granularity then 4 disk accesses per each hour will be required. And so forth for finer granularity. To overcome this

problem, we have suggested an improvement to this approach. We can combine two levels together in one level as in Figure 3.3. We combined years and months together in one level. Therefore, reducing the disk access one access for every search. But as event increases, more blocks for these combined levels will be used. Consequently; either binary search will be implemented to search through these blocks which is worse than the first approach in Figure 3.2, or we could go back to the previous approach and add one extra level as in Figure 3.2.

3.2.3 Hashed-Cluster Temporal Index

In order to minimize levels of access, a better solution is needed than the solution that is illustrated in Figure 3.3. As we know, universal timing are divided into 12 months a year, 28-31 days a month, 24 hours a day, ... etc. This timing scheme is adequate to represent all time stamps, valid time stamp, transaction time stamps, and bi-temporal time stamps.

Universal Timing (UT) posses a predictable rhythm. Hence; this property can be used to find the location of the indexed record more efficiently. This indexing scheme will be named after the used concepts within the index structure. It will be named Hashed-cluster temporal index. In this index a hash function can be used to map the used granularities through an intermediate one level of index file as shown in Figure 3.4.

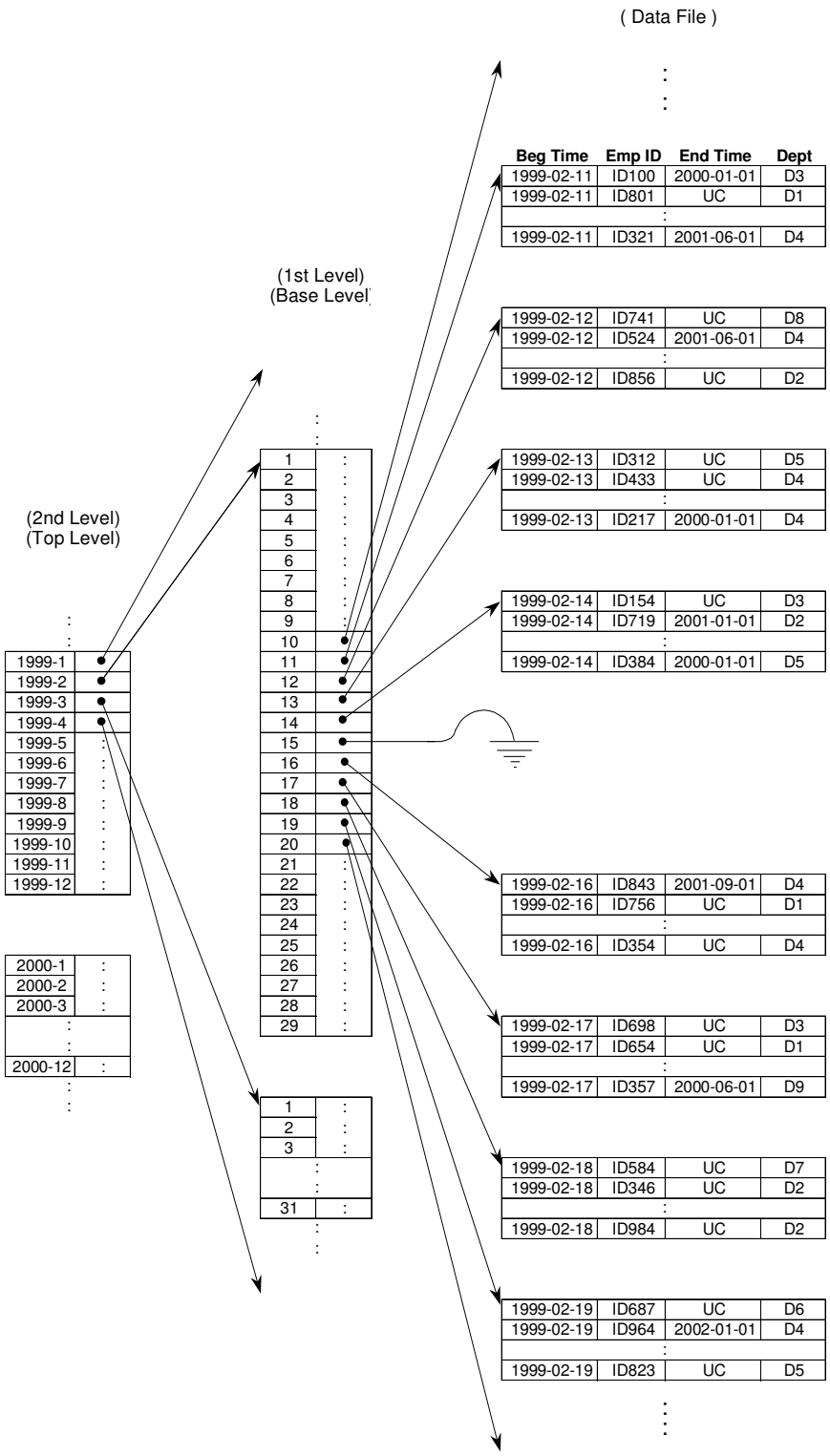


Figure 3.3. Modified multi-level index

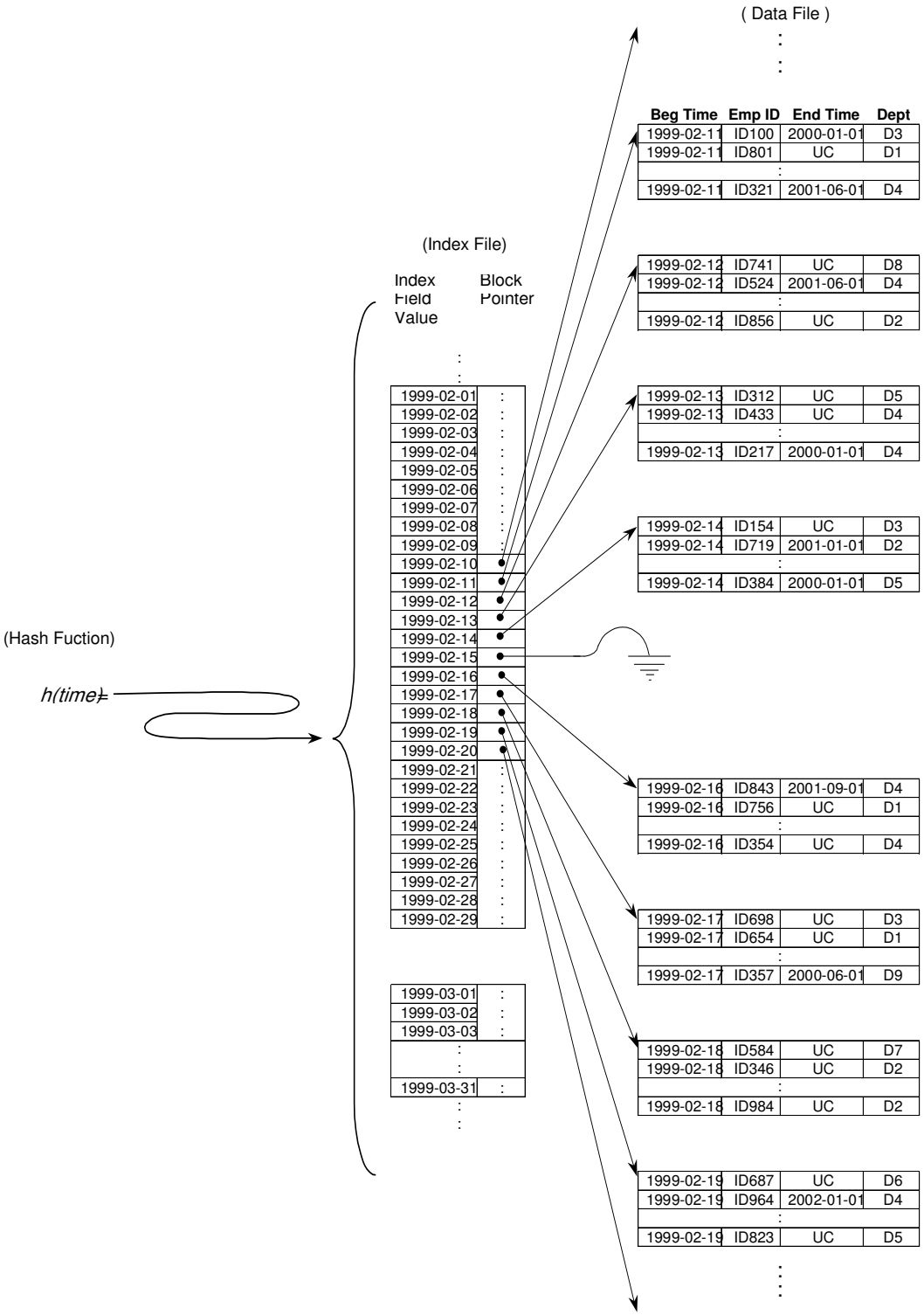


Figure 3.4. Hash-cluster index

Since the main index key is “starting time”, then the first “starting time” stamp of *each relation* in a temporal relational database will be recorded in the data dictionary of the database. Used granularity will be defined too. Plus other regular meta data about the database as shown in Table 3.10 . All of these meta data will be included in the database dictionary.

Table 3.10. Temporal database dictionary (Meta data)

Relation Name	Granularity	Starting Time	No of Records	Etc
Department	Day	1999-01-01	2471	...
Operations	Hour	1999-03-26-17	54826	...
.	.	.	.	.
.	.	.	.	.

As can be seen from Figure 3.4, the hash function takes time stamp identification as a parameter and returns its exact block address through a fixed intermediate index file. As an example. Suppose that we need to search for records that start time stamp is 1999-4-23 in Department relation. We can find its location by using a function that returns the coordinate of the time stamp “1999-4-23”. From the data dictionary in Table 310 we can determine the starting point from which to start counting for the position of this date in the index file.

Hence,

Day_Coordinat_Function("1999-04-23") = 31+28+31+23=113

Therefore,

Exact index position address = Start memory address of index file + 113

Functions that are used to extract the coordinate will depend on the granularity definition for the desired relation as specified in the database dictionary. We can use the coordinate of "Start Time" stamp and the "Starting Time" for the specified relation in order to find its exact pointer location that will lead us to the exact block address as shown above.

The approach used in Figure 3.2 will require 3 access for every searched record if the used granularity is "Day". It will require 4 extra disk access if the used granularity is "Hour" and so forth. With the Hashed-cluster approach of Figure 3.4, the number of disk access will always be only one access to the disk regardless of defined granularity, even if we used microsecond granularity.

Algorithm 3.2 illustrate the search algorithm for a record in the data file using the hash function that is used in the context of hashed-cluster index.

Algorithm 3.2. Searching in Hashed-cluster temporal index

```
        Hash_Function ("Start Time Stamp") Pointer

        Begin
Step 1:     Base_Position ← Relation "Starting Time"
Step 2:     Record_Coordinate ←X_Coordinate_Funtion(Start -
                Time Stamp)
Step 3:     Actual_Position ← Base_Postion + Record_Coordinate
Step 4:     Block_Pointer(Actual_Position) ← Record_Pointer
Step 5:     Hash_Function ← Record_Pointer
        End
```

"Base_Position" represent the starting memory address from which to start counting. "Base_Position" can be found in the database dictionary, or it can be determined upon downloading the index file from the hard disk to the RAM. "X_Coordinat_Function" as we said before will be used according to the granularity defined in the data dictionary. The "X" in the function represents the granularity. If the granularity is "Day", then we use "Day_Coordinate_Function". If the used granularity is "Hour" then we used "Hour_Coordinate_Function" and so on for finer granularity. Through the intermediate index file, the exact block position (Actual position) can be found by adding the base memory address to the record coordinate.

Since the cost of the processing time is negligible. And the disk access time is the dominant cost factor in large database [Ozsu, 1999] and [Silberschatz, 1997]. This indexing scheme (Hashed-cluster index) takes the burden of disk access and passes it to processing time. Hence, tremendous time is saved in searching.

The Hashed-cluster index methodology can be used for continuous events database where there are always events takes place all the time, so no null pointers is used.

In case of having discrete events, we could improve this index methodology to have non-dense index file.

Quasi-append only databases are recommended to use this index for.

This approach takes “Start Time” stamp as primary cluster index. More index keys can be added to primary “Star Time” to form second, third, and forth multiple-keys primary index. These keys could be “End Time” stamp, surrogate attribute, and temporal attribute fields. Which will help in implementing various types the joining algorithms according to our needs. To add this multiple-key clustered index capabilities to the database we have to consider each type of the temporal database separately. The two main types of temporal databases that will be discussed here are roll back databases which depend on transaction time

stamping, and historical databases, which depends on valid time stamping. Bi-temporal databases are defined within the context of roll-back and historical databases, because it's a combination of both. In the next three sub-sections we will discuss how multiple-keys primary clustered index can be implemented for historical and rollback temporal databases, in addition to ending time for rollback transaction times and historical valid time.

3.2.3.1 Transaction start time in rollback databases

Rollback database uses transaction start time and transaction end time. Since the nature of transaction times update happens naturally in ascending order, then primary hashed-clustered index as explained above can be used to index these types of databases.

3.2.3.2 Valid start time in historical databases

Historical relational databases uses valid start time and valid end time. The file structure of the historical databases would be hard to be kept in contiguous memory location. Therefore, a secondary clustered index will be used instead of primary clustered index but with the same hash function and same intermediate index file. Unused intermediate pointers can have reserved blank pointers that can be used upon request. Each

group of records that carry same valid start time have to be kept together in contiguous memory location in one cluster but clusters does not have to be ordered according to its valid start time in contiguous locations. And this is what we mean by secondary clustered index. Nature of valid time entry is unpredictable which necessitate the use of this method. This method requires regular disk de-fragmentation and reorganization of records.

3.2.3.3 Valid and transaction end times in rollback and historical databases.

Valid end time in historical databases, and transaction end time in rollback databases can be added normally during updates as follows: Transaction end time will be added automatically by the system and a new tuple for the same key is created. And that happens when the related tuple is updated.

Valid end time can be either one of the following two cases. First case where valid end time is left blank because it is unknown. (e.g. an employee termination date). This type has to be updated manually upon user entry. Second case where valid end time is known ahead of time. In this case it is entered upon the tuple creation. Old records will be closed by filling it's valid end time. And at the same time a new tuple will be

created. The new tuple valid start time is greater in one unit time than valid end time for the old closed tuple. Bear in mind, that upon updating tuples by entering its end time, the records have to be reorganized and sorted by end time or any other desired key according to our needs, as a second or more sorting key/s. Of course start time will be the first sorting key. And this would be possible if we use fixed length records. This would be a simple cheap operation since no more extra space would be required. This organization comes in handy when sorting is required to fit specific join algorithm in query optimization operation. In this case sorting will be implemented in place.

Based in this discussion we can see that transaction time is suitable for continuous events. And valid time is suitable for discrete events. Transaction time is suitable for events that are recorded as it happen and continue to exist until further notice. But valid time is suitable to record events that might be recorded in retroactive and proactive basis.

3.2.4 Temporal Relational Databases Operation

As in the regular relational databases. The temporal relational databases operations are insert, delete, and update. Table 3.11 summarize suggestion to the use of these operations in temporal databases index.

Table 3.11. Possible operation on temporal database index

Operation		**Historical DB (Valid Time)**	**Roll Back DB (Transaction Time)**
Append		To related cluster or overflow	To end of relation
Delete		Marked out as deleted	Marked out as deleted
Update	Correction	if start time, delete and append (Normally). If end time, & others reorganize related cluster.	If start time, delete and append. (Overflow) If end time, & others reorganize related cluster.
	Entering End time	Reorganize related cluster	Reorganize related cluster

These restrictions have to be built in within the Temporal Relational Database Management System. Because temporal relations inside a relational database have to be maintain and sorted as explained earlier in section 3.1 in order to use its temporal dimension efficiently according to Hashed-cluster index. And follows we recap the methodology to do so in transaction roll back database.

One batch of clustered records that contain the most current records, which belong to a specified start time stamp, are being dealt with in a temporary file (Temporal Buffer Table TBT). This batch of clustered records will be added to the main related relations within the database upon the beginning of the next indexed time stamp that are defined according to used granularity. We add these clustered records into its

contiguous memory location at the end of the related relation. But before we add these clustered records and start the next temporary time stamp file. We may sort the records according to “End Time” stamp or any other one or more key/s. In this way we have “Start Time” and “End Time” as a multiple primary index which will helps to facilitate different join algorithms.

This hashed-cluster index lends itself to facilitate joining between incompatible time granularity for either index methodology as in Figure 3.2, or as in Figure 3.4. In one condition, that is, granularity have to be generalized but not specialized as depicted in Figure 3.5.

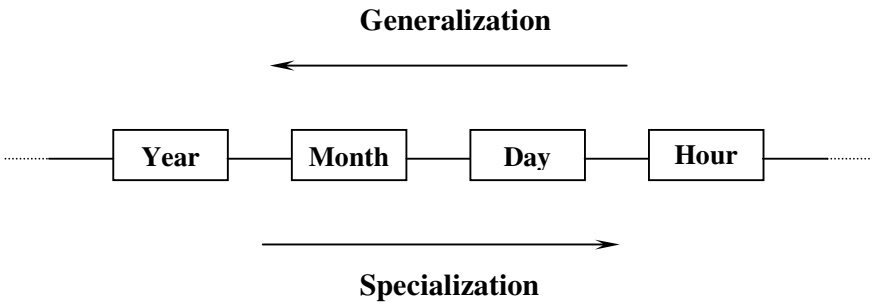


Figure 3.5. Generalization and specialization in time granularities

More details about dealing with incompatible granularities can be found in [Dyreson, 1994].

3.3 JOIN PROCESSING

3.3.1 Overview

In previous sections we have discussed important factors in query optimization. That is, temporal relational databases schema and indexing. The third most important factor is join processing, which will be discussed in this section. In order to represent temporal data in this section we have used the time interval representation [Toman, 1996] and [Toshiyuki, 1999] as will be shown in the next sections.

To represent our findings, we have used the following schema, which contain employee relation as in Table 3.12. This relation is in the snapshot relation of the Employee relation.

Table 3.12. Employee snap-shot relation

Emp_ID	Name	DOB	Start	Salary	Sal_Ts	Dept	Dept_Ts
01	Ahmad	01/01/1960	1	100	51	3	46
02	Ali	02/02/1961	21	110	71	2	79
03	Raed	03/03/1960	51	85	116	5	141
04	Mona	04/04/1970	91	120	146	1	148
05	Mai	05/05/1980	151	95	191	3	181
06	Khalid	06/06/1980	161	80	191	5	198
07	Sami	07/07/1985	111	100	196	2	191
08	Osamah	08/08/1961	90	120	197	2	191
09	Jamal	09/09/1962	146	135	191	3	194

As can be seen form the “Employee” snapshot relation in Table 3.12 above. There are three non-temporal attributes, which will stay in this

relation, they are Name, DOB, and Start attributes. In addition, there are two temporal attributes, they are Salary and Dept (Department). The last snapshot value of these temporal attributes are kept within "Employee" snapshot relation with a hidden field next to each one (Shaded in Gray) that point to the last value in the temporal relations that are related to "Employee" relation. Temporal relations that have emerged from "Employee" relations are shown in Tables 3.13.

Table 3.13a. Temporal employee - department relation

Emp_ID	Ts	Te	Dept
01	1	15	1
01	16	30	3
01	31	45	1
01	46	60	3
02	21	60	2
02	61	65	3
02	66	69	1
.	.	.	.
.	.	.	.

Table 3.13b. Temporal employee - salary relation

Emp_ID	Ts	Te	Salary
01	1	10	50
01	11	15	55
01	16	20	60
01	21	30	70
01	31	40	75
01	41	45	80
01	46	50	85
.	.	.	.
.	.	.	.

The two temporal relations in Tables 3.13 will be used through out this section to demonstrate the joining algorithms.

Following are terms that have been used and their representation:

Term	Representation
Surrogate	: "S", Key attributes (Emp_ID)
Temporal attribute	: "A",Temporal attributes (Salary and Department)
Time attributes	: Start time stamp "Ts" and end time stamp "Te"

The used temporal relations are in first temporal normal form TNF as specified by Tansel [1993] which states that a relation is in Temporal Normal Form (TNF) if and only if it is in BCNF (Boyce-Codd Normal Form) and there exist no temporal dependencies among its none-key attributes.

Life span of a relation is identified to start with its earliest start time tuple, Start-Life-Span(R)=$MIN_r\{r\ (\text{Ts})\}$. And it ends with the latest end time End-Life-Span(R)=$MAX_r\{r\ (\text{Te})\}$.

There are many types of temporal joins [Gunadhi, 1990]. In this section, we show a special interest in the most famous kind of temporal join and that is Time-intersection Equi-join. In this type of join, two tuples from two different temporal relations can be joined together if their surrogate attribute are equal and their time interval intersects. This type of temporal join will be used in this section in the following sections. The following conventions are used through out this section. Let R_1 and R_2 be the joining relations. r_1 and r_2 represent tuples from R_1 and R_2, respectively. *bfr* is the blocking factor.

With regards to time stamps in temporal relational databases, there are tow types of temporal relations. First type posses a continuity nature. In this type if a tuple is closed by filling in the “Te1”, a new tuple have to be created at the same time. And it will carry the new temporal attribute value that causes the change. Its “Ts2” will be equal to old “Te1” for the old tuple plus one unit of used time granularity. That is:

$$Ts2 = Te1 + 1$$

As an example, in “Salary” temporal relation, the salary of an employee will remain valid until s/he gets a raise, which will start right after the ending time of the old salary interval. There can not be an interval without a salary.

The second type posses a discrete nature. Closing a tuple does not have to initiate a new tuple. As an example, suppose we have a training temporal relation for the “Employee” relation. An employee might be assigned for a training course for a specific period of time. When the training interval ends. S/he does not have to have a second training course right after it. In this type of temporal relation, events are discrete. While for the first type, events are continuous. When designing algorithms for joining temporal relations, one have to take in consideration this fact in order to optimize the join to maximum possible limit.

In this section the first 3 algorithms in the next sections, that is algorithm 3.3, 3.4, and 3.5, are assumed to have continuous time temporal relation for both joined relations.

Algorithms 3.4, 3.5, and 3.6 below, are supported by the quasi-append databases that we have suggested to use in Section 3.2.

Merge-join and nested loop join will be used for simplicity of the algorithms. As hash join is expected to imply a complex algorithm, it will not be used.

The important question that may arise, do we really need algorithm that may not be optimal?

As we know that an algorithm might not be optimal at one stage. Nevertheless; the overall plan may constitute the optimal path. More clearly, the nonoptimal algorithm may lead to optimal subsequent joins that make up for the overhead caused by an earlier bottleneck process. As Silberschatz [1997] stated "To choose the best overall algorithm, we must consider even nonoptimal algorithms for individual operations".

3.3.2 Join Using S, and Ts

To understand how we can design an optimal algorithm we have used two relations from the above database. These two relations are "Department" (Table 3.14) and "Salary" (Table 3.15)

Table 3.14. Department relation

Emp_ID	Ts	Te	Dept
01	1	15	1
01	16	30	3
01	31	45	1
01	46	60	3
02	21	60	2
02	61	65	3
02	66	69	1
02	70	74	2
02	75	78	5
02	79	80	2
03	51	60	5
03	61	90	4
03	91	140	1
03	141	150	5
04	91	95	1
04	96	100	2
04	101	104	5
04	105	110	3
04	111	120	1
04	121	126	2
04	127	130	4
04	131	134	1
04	135	138	4
04	139	142	5
04	143	147	3
04	148	150	1
05	151	160	4
05	161	180	5
05	181	200	3
06	161	163	1
06	164	169	2
06	170	174	4
06	175	179	3
06	180	181	1
06	182	184	5
06	185	189	3
06	190	194	1
06	195	197	2
06	198	200	5
07	111	115	2
07	116	120	1
07	121	125	5
07	126	130	1
07	131	135	5
07	136	140	3
07	141	150	4
07	151	160	1
07	161	170	4
07	171	180	2
07	181	190	4
07	191	200	2
08	90	110	2
08	111	130	1
08	131	170	3
08	171	190	1
08	191	200	2
09	146	150	5
09	151	155	1
09	156	159	4
09	160	168	5
09	169	170	3
09	171	181	1
09	182	193	2
09	194	200	3

Table 3.15. Salary relation

Emp_ID	Ts	Te	Salary
01	1	10	50
01	11	15	55
01	16	20	60
01	21	30	70
01	31	40	75
01	41	45	80
01	46	50	85
01	51	60	100
02	21	30	60
02	31	40	65
02	41	50	70
02	51	55	75
02	56	60	85
02	61	65	90
02	66	70	100
02	71	80	110
03	51	60	45
03	61	65	49
03	66	70	53
03	71	80	58
03	81	85	62
03	86	90	65
03	91	95	68
03	96	105	70
03	106	110	75
03	111	115	80
03	116	150	85
04	91	95	50
04	96	100	55
04	101	105	60
04	106	110	65
04	111	115	70
04	116	120	75
04	121	125	80
04	126	130	85
04	131	135	90
04	136	140	100
04	141	145	110
04	146	150	120
05	151	160	60
05	161	170	70
05	171	180	80
05	181	190	90
05	191	200	95
06	161	170	60
06	171	180	70
06	181	190	75
06	191	200	80
07	111	120	50
07	121	130	55
07	131	135	60
07	136	140	65
07	141	150	70
07	151	160	75
07	161	175	80
07	176	185	85
07	186	195	95
07	196	200	100
08	90	100	40
08	101	110	45
08	111	115	50
08	116	125	56
08	126	130	70
08	131	140	73
08	141	150	75
08	151	165	78
08	166	170	80
08	171	173	85
08	174	175	88
08	176	177	90
08	178	181	93
08	182	184	95
08	185	186	97
08	187	189	99
08	190	192	100
08	193	193	110
08	194	196	115
08	197	200	120
09	146	150	70
09	151	154	85
09	155	160	100
09	161	170	110
09	171	180	125
09	181	190	130
09	191	200	135

Both relations are sorted by surrogate attribute “Emp_ID” and start time stamp “Ts”. Figure 3.6 illustrate a chart for “Department” relation. And Figure 3.7 illustrate a chart for “Salary” relation.

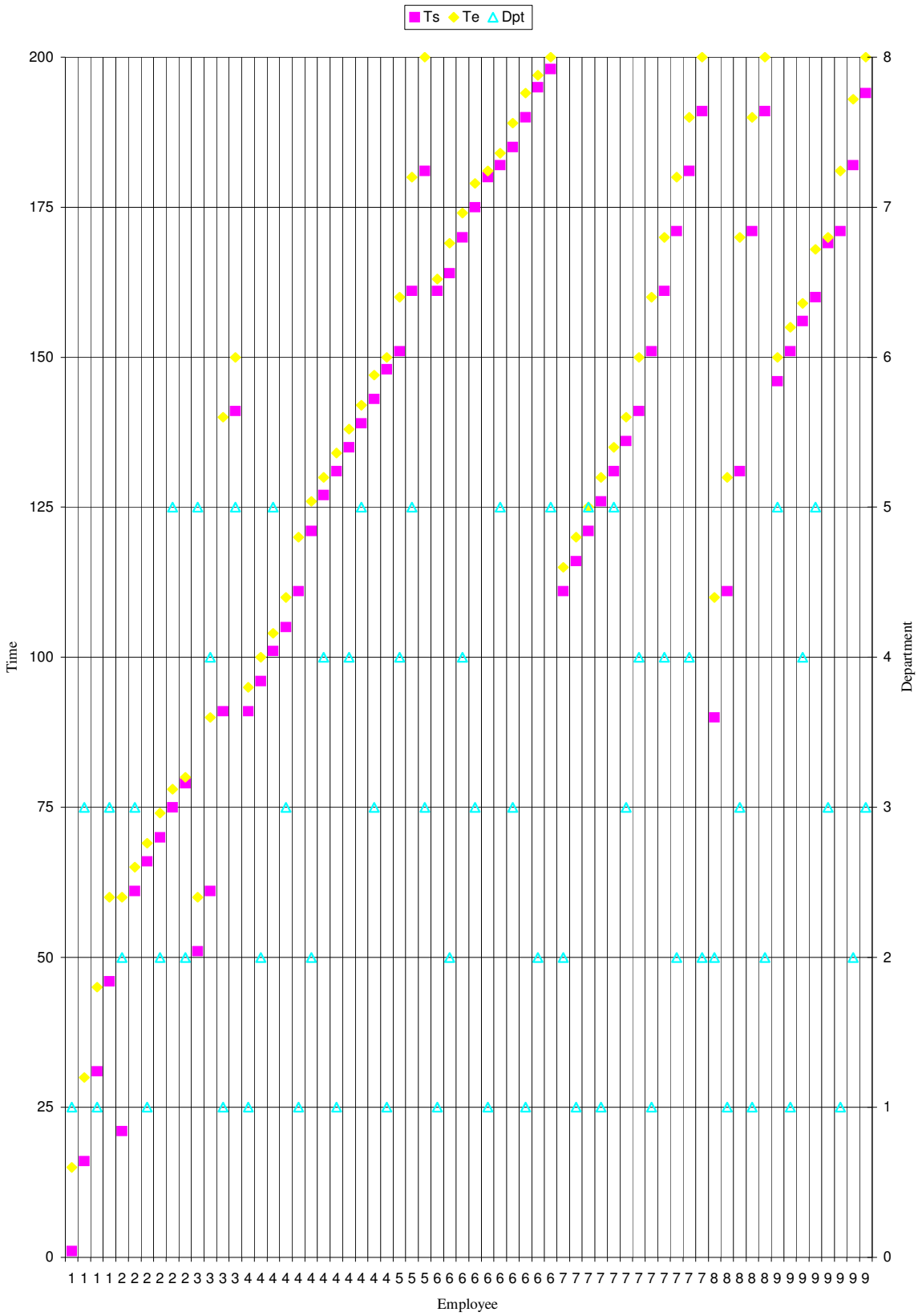


Figure 3.6. Department relation chart, sorted by S, & Ts

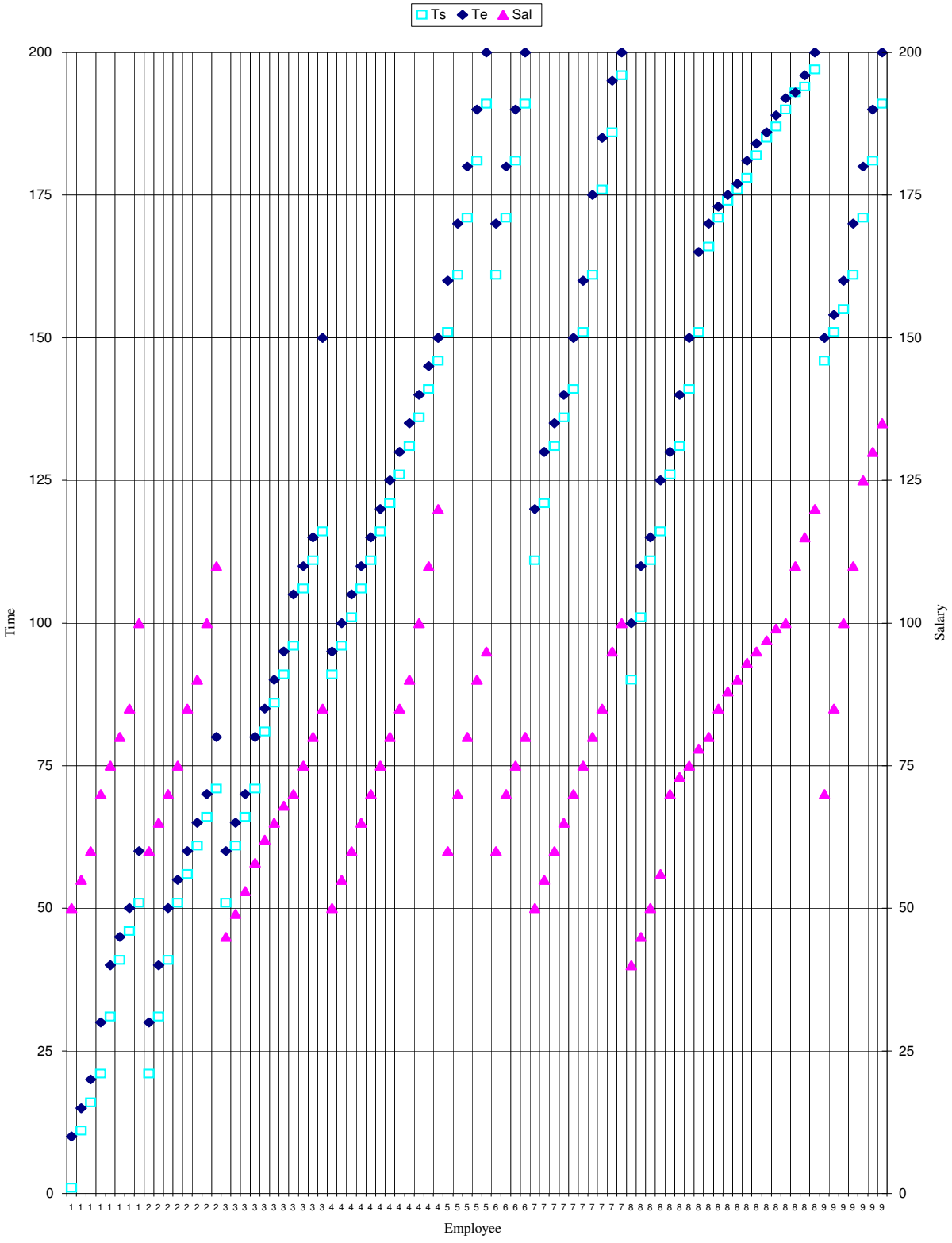


Figure 3.7 Salary relation chart, sorted by S & Ts

As we can see from the department and salary relation charts. The employee time start for the first event for each employee matches. Also, the ending time for the last event matches too for the each employee. Within the complete time frame for each employee the time slices segments either starts, end, or overlap with each other's. The overlaps could be in either direction but certainly, time events for each employee ends at the same time. Based on these properties we have designed algorithm 3.3 to join both relations.

Algorithm 3.3 below illustrates the process of joining temporal relation that are of continuous nature. And based on temporal relations that are sorted by surrogate and starting time stamp.

Algorithm 3.3. TE-join for relations that are sorted by S, and Ts. Where the link is based on time intersection plus surrogate attribute.

Step 1 : Read *r1* from R_1 and *r2* from R_2

Step 2 : Repeat until $R_1.eof$ and $R_2.eof$

Three cases to consider:

1- $Te_1 < Te_2$ Produce output tuple,
Identify uncovered portion of *r2* as *r2*,
Read next *r1*.

2- $Te_1 = Te_2$ Produce output tuple,
Read next *r1* and *r2*.

3- $Te_1 > Te_2$ Produce output tuple,
Identify uncovered portion of *r1* as *r1*,
Read next *r2*.

Cost:

Since for each tuple in one side of the join relation there is a matching tuple in the other side of the join relations, this type of join perform the most optimize join between relation in join processing of temporal relational databases.

Sort merge can be used for this kind of join. Both relations will be scanned only once. Therefore, the cost of this join will be minimal.

$$C_{5.1} = b_{r1} + b_{r2}$$

Where b is the number of blocks. The algorithm has been implemented in the computer. The source code for the implementation can be seen in Annex 1. Microsoft Access has been used to write the codes. The output of the implementation is illustrated in Table 3.16.

Table 3.16. Output of algorithm 3.3.

Emp_ID	Ts	Te	Dept	Salary
01	1	10	1	50
01	11	15	1	55
01	16	20	3	60
01	21	30	3	70
01	31	40	1	75
01	41	45	1	80
01	46	50	3	85
01	51	60	3	100
02	21	30	2	60
02	31	40	2	65
02	41	50	2	70
02	51	55	2	75
02	56	60	2	85
02	61	65	3	90
02	66	69	1	100
02	70	70	2	100
02	71	74	2	110
02	75	78	5	110
02	79	80	2	110
03	51	60	5	45
03	61	65	4	49
03	66	70	4	53
03	71	80	4	58
03	81	85	4	62
03	86	90	4	65
03	91	95	1	68
03	96	105	1	70
03	106	110	1	75
03	111	115	1	80
03	116	140	1	85
03	141	150	5	85
04	91	95	1	50
04	96	100	2	55
04	101	104	5	60
04	105	105	3	60
04	106	110	3	65
04	111	115	1	70
04	116	120	1	75
04	121	125	2	80
04	126	126	2	85
04	127	130	4	85
04	131	134	1	90
04	135	135	4	90
04	136	138	4	100
04	136	140	5	100
04	141	142	5	110
04	143	145	3	110
04	146	147	3	120
04	148	150	1	120
05	151	160	4	60
05	161	170	5	70
05	171	180	5	80
05	181	190	3	90
05	191	200	3	95
06	161	163	1	60
06	164	169	2	60
06	170	170	4	60
06	171	174	4	70
06	175	179	3	70
06	180	180	1	70
06	181	181	1	75
06	182	184	5	75
06	185	189	3	75
06	190	190	1	75
06	191	194	1	80
06	195	197	2	80
06	198	200	5	80
07	111	115	2	50
07	116	120	1	50
07	121	125	5	55
07	126	130	1	55
07	131	135	5	60
07	136	140	3	65
07	141	150	4	70
07	151	160	1	75
07	161	170	4	80
07	171	175	2	80
07	176	180	2	85
07	181	185	4	85
07	186	190	4	95
07	191	195	2	95
07	196	200	2	100
08	90	100	2	40
08	101	110	2	45
08	111	115	1	50
08	116	125	1	56
08	126	130	1	70
08	131	140	3	73
08	141	150	3	75
08	151	165	3	78
08	166	170	3	80
08	171	173	1	85
08	174	175	1	88
08	176	177	1	90
08	178	181	1	93
08	182	184	1	95
08	185	186	1	97
08	187	189	1	99
08	190	190	1	100
08	191	192	2	100
08	193	193	2	110
08	194	196	2	115
08	197	200	2	120
09	146	150	5	70
09	151	154	1	85
09	155	155	1	100
09	156	159	4	100
09	160	160	5	100
09	161	168	5	110
09	169	170	3	110
09	171	180	1	125
09	181	181	1	130
09	182	190	2	130
09	191	193	2	135
09	194	200	3	135

3.3.3 Join Using Ts, and S

There are special cases were relations could be sorted by start time stamp "Ts" and surrogate attribute "S". These relations possess different characteristics than those explained in section 3.3.2. To illustrate these characteristics we have used the same relation as in section 3.3.2. But we have sorted them on Ts and S attributes. Table 3.17 illustrate "Department" relation sorted on Ts & S attributes. Table 3.18 illustrate the "Salary" relation sorted by Ts & S.

Table 3.17. Department relation sorted by Ts & S

Ts	Emp_ID	Te	Dept
1	01	15	1
16	01	30	3
21	02	60	2
31	01	45	1
46	01	60	3
51	03	60	5
61	02	65	3
61	03	90	4
66	02	69	1
70	02	74	2
75	02	78	5
79	02	80	2
90	08	110	2
91	03	140	1
91	04	95	1
96	04	100	2
101	04	104	5
105	04	110	3
111	04	120	1
111	07	115	2
111	08	130	1
116	07	120	1
121	04	126	2
121	07	125	5
126	07	130	1
127	04	130	4
131	04	134	1
131	07	135	5
131	08	170	3
135	04	138	4
136	07	140	3
139	04	142	5
141	03	150	5
141	07	150	4
143	04	147	3
146	09	150	5
148	04	150	1
151	05	160	4
151	07	160	1
151	09	155	1
156	09	159	4
160	09	168	5
161	05	180	5
161	06	163	1
161	07	170	4
164	06	169	2
169	09	170	3
170	06	174	4
171	07	180	2
171	08	190	1
171	09	181	1
175	06	179	3
180	06	181	1
181	05	200	3
181	07	190	4
182	06	184	5
182	09	193	2
185	06	189	3
190	06	194	1
191	07	200	2
191	08	200	2
194	09	200	3
195	06	197	2
198	06	200	5

Table 3.18. Salary relation sorted by Ts & S

Ts	Emp_ID	Te	Salary
1	01	10	50
11	01	15	55
16	01	20	60
21	01	30	70
21	02	30	60
31	01	40	75
31	02	40	65
41	01	45	80
41	02	50	70
46	01	50	85
51	01	60	100
51	02	55	75
51	03	60	45
56	02	60	85
61	02	65	90
61	03	65	49
66	02	70	100
66	03	70	53
71	02	80	110
71	03	80	58
81	03	85	62
86	03	90	65
90	08	100	40
91	03	95	68
91	04	95	50
96	03	105	70
96	04	100	55
101	04	105	60

101	08	110	45
106	03	110	75
106	04	110	65
111	03	115	80
111	04	115	70
111	07	120	50
111	08	115	50
116	03	150	85
116	04	120	75
116	08	125	56
121	04	125	80
121	07	130	55
126	04	130	85
126	08	130	70
131	04	135	90
131	07	135	60
131	08	140	73
136	04	140	100
136	07	140	65
141	04	145	110
141	07	150	70
141	08	150	75
146	04	150	120
146	09	150	70
151	05	160	60
151	07	160	75
151	08	165	78
151	09	154	85
155	09	160	100
161	05	170	70

161	06	170	60
161	07	175	80
161	09	170	110
166	08	170	80
171	05	180	80
171	06	180	70
171	08	173	85
171	09	180	125
174	08	175	88
176	07	185	85
176	08	177	90
178	08	181	93
181	05	190	90
181	06	190	75
181	09	190	130
182	08	184	95
185	08	186	97
186	07	195	95
187	08	189	99
190	08	192	100
191	05	200	95
191	06	200	80
191	09	200	135
193	08	193	110
194	08	196	115
196	07	200	100
197	08	200	120

Both relations above have been charted. Figure 3.8 illustrate a chart for "Department" relation and Figure 3.9 illustrate a chart for "Salary" relation.

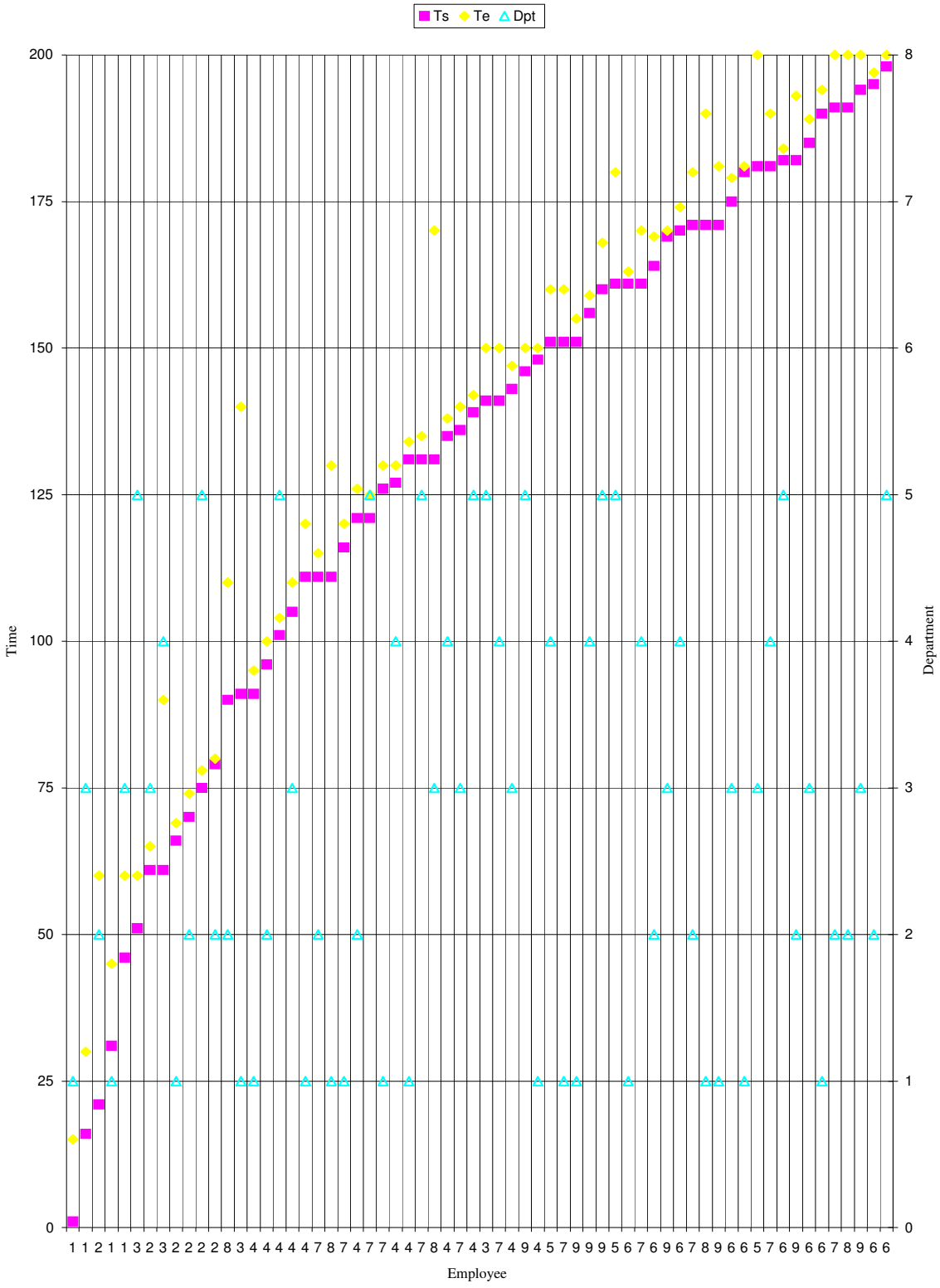


Figure 3.8. Department relation chart, sorted by Ts & S

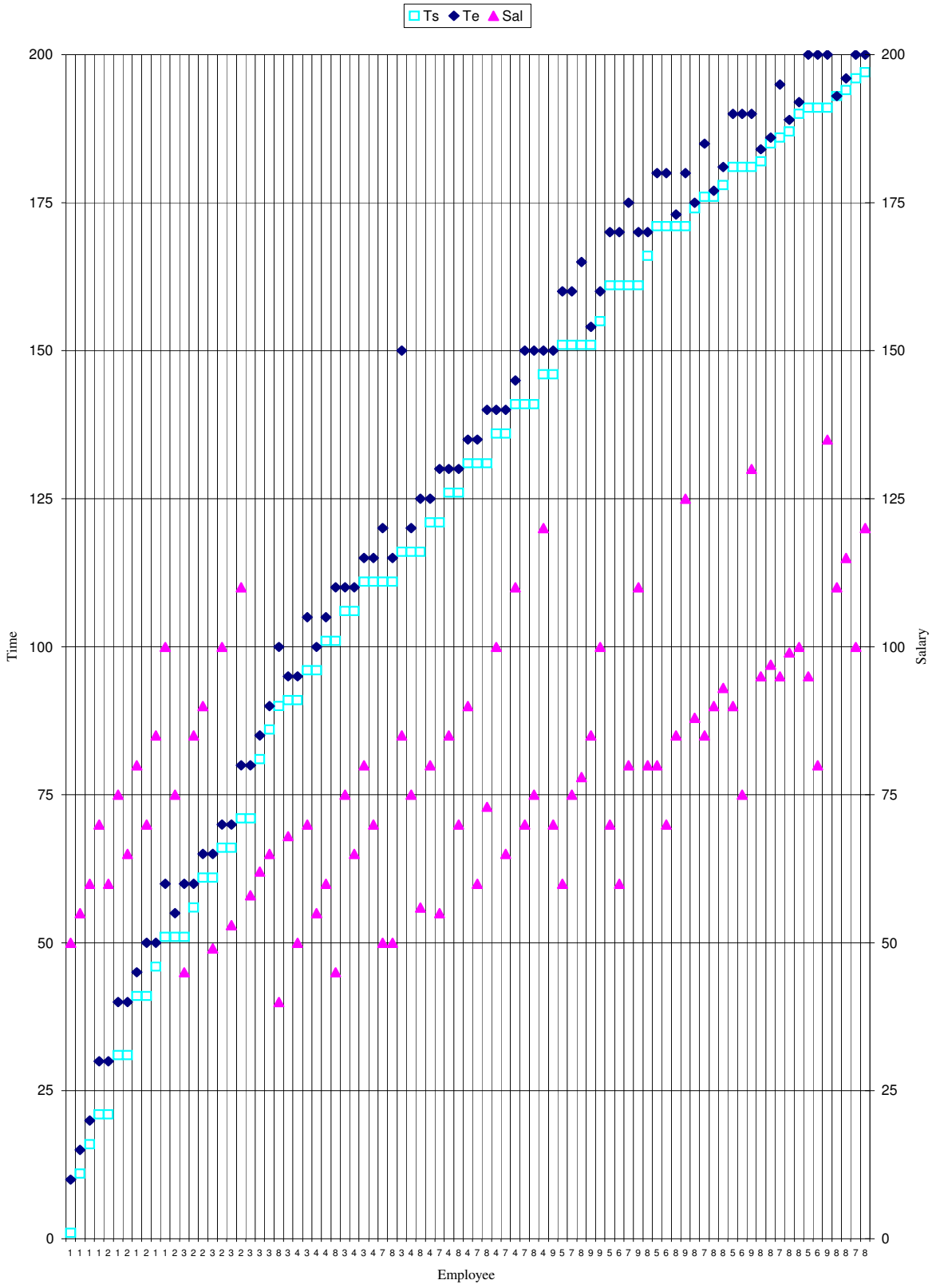


Figure 3.9. Salary relation chart, sorted by Ts & S

Since both relations are sorted on Ts & S, then single loop join methodology will be used. Any relation could be used as outer relation. As can be noted from the chart that for each time slices in “Department” relation there are two chances. First chance is to find a corresponding tuple in “Salary” relation. Second chance is not to find the exact matching tuple. Fortunately, In this case the desired tuple will be found close to where the searching pointer stops using the same available index due to unsuccessful hit.

To explain this point more clearly let us takes an example from the chart. Assume that the current tuple at “Department” relation is where Ts=70, Te=74, Emp_ID=02, and Dept=2. As we know, we assume that the both relation are indexed in a primary key that contain Ts & S . Therefore, The search in “Salary” relation will be for a tuple that has “70 02” value as primary key and the searching pointer will stop in “71 02” tuple and will return a message to the user indicating that the record is not found. Since the “Salary” tuple at which the searching pointer stops, start with 71 as it’s “Ts” value, records in ascending order including this record will not be our goal. Because at least the “70” value or more will not be covered. And the coverage of this value is therefore guaranteed to be found in the nearby preceding tuples, because both relations are of continuous nature.

Based on these facts. This algorithm, in its worst cases, will search for a small portion of the relation for the desired tuple (assuming) normal distribution. Algorithm 3.4 has been designed to benefit from these characteristics.

Algorithm 3.4. TE-join for relations that are sorted by Ts, and S. Where the link is based on time intersection plus surrogate attribute.

Step 1 : Select next *r1* from R_1 until eof
Step 2 : Seek for *r1* in R_2, if not found then step 4
Step 3 : Two cases to consider:
1- $Te_1 <= Te_2$, Produce output tuple,
Go to step 1
2- $Te_1 > Te_2$, Produce output tuple.
Identify uncovered portion as *r1*,
Go to step 2
Step 4 : seek for *r1* in R_2 such that $Ts_2 < Ts_1 <= Te_2$, go to step 3

This algorithm has been implemented in the computer. Annex 1 contains the source code for the implementation.

As for algorithm 3.3, Microsoft Access has been used to write the source code. The output of the implementation is illustrated in Table 3.19.

Cost:

In step 2, if the tuple were found then it would cost us one disk access. If the desired tuple is not found then step 4 will be implemented, and we calculate the cost as follows. Assume that the searching pointer (as explained earlier) stops at tuple *r*21 and the correct matching tuple is *r*22. The distance between *r*21 and *r*22 is *D*. The algorithm have to scan *D* in order to reach *r*22 and *Dr* tuples will be scanned. Number of blocks for these tuples will be added as disk access in order to calculate the cost. So, the cost will be:

$$C_{5.2} = b_{r1} + (| r_1 | * (D_{r2} / bfr))$$

Outer relation should be the relation with less number of records in order to increase the number of successful seek (hits) and at the same time reduce the number of unsuccessful seek (hits). And therefore to achieve better performance.

Table 3.19. Output of algorithm 3.4.

Ts	Emp_ID	Te	Dept	Salary
1	01	10	1	50
11	01	15	1	55
16	01	20	3	60
21	01	30	3	70
21	02	30	2	60
31	01	40	1	75
31	02	40	2	65
41	01	45	1	80
41	02	50	2	70
46	01	50	3	85
51	01	60	3	100
51	02	55	2	75
51	03	60	5	45
56	02	60	2	85
61	02	65	3	90
61	03	65	4	49
66	02	69	1	100
66	03	70	4	53
70	02	70	2	100
71	02	74	2	110
71	03	80	4	58
75	02	78	5	110
79	02	80	2	110
81	03	85	4	62
86	03	90	4	65
90	08	100	2	40
91	03	95	1	68
91	04	95	1	50
96	03	105	1	70
96	04	100	2	55
101	04	104	5	60
101	08	110	2	45
105	04	105	3	60
106	03	110	1	75
106	04	110	3	65
111	03	115	1	80
111	04	115	1	70
111	07	115	2	50
111	08	115	1	50
116	03	140	1	85
116	04	120	1	75
116	07	120	1	50
116	08	125	1	56
121	04	125	2	80
121	07	125	5	55
126	04	126	2	85
126	07	130	1	55
126	08	130	1	70
127	04	130	4	85
131	04	134	1	90
131	07	135	5	60
131	08	140	3	73
135	04	135	4	90
136	04	138	4	100
136	04	140	5	100
136	07	140	3	65
141	03	150	5	85
141	04	142	5	110
141	07	150	4	70
141	08	150	3	75
143	04	145	3	110
146	04	147	3	120
146	09	150	5	70
148	04	150	1	120
151	05	160	4	60
151	07	160	1	75
151	08	165	3	78
151	09	154	1	85
155	09	155	1	100
156	09	159	4	100
160	09	160	5	100
161	05	170	5	70
161	06	163	1	60
161	07	170	4	80
161	09	168	5	110
164	06	169	2	60
166	08	170	3	80
169	09	170	3	110
170	06	170	4	60
171	05	180	5	80
171	06	174	4	70
171	07	175	2	80
171	08	173	1	85
171	09	180	1	125
174	08	175	1	88
175	06	179	3	70
176	07	180	2	85
176	08	177	1	90
178	08	181	1	93
180	06	180	1	70
181	05	190	3	90
181	06	181	1	75
181	07	185	4	85
181	09	181	1	130
182	06	184	5	75
182	08	184	1	95
182	09	190	2	130
185	06	189	3	75
185	08	186	1	97
186	07	190	4	95
187	08	189	1	99
190	06	190	1	75
190	08	190	1	100
191	05	200	3	95
191	06	194	1	80
191	07	195	2	95
191	08	192	2	100
191	09	193	2	135
193	08	193	2	110
194	08	196	2	115
194	09	200	3	135
195	06	197	2	80
196	07	200	2	100
197	08	200	2	120
198	06	200	5	80

3.3.4 Join Using Ts, Te, and S

This kind of join is recommended for time dependent events in which an event would create tuples in two or more relation. These tuples would have the same Ts, Te, and S values. The difference in these tuples would be the value of "A" attribute that would be related to the underlying relation.

As an example to explain the behavior of this kind of joins that are based in Ts, Te, and S as join attributes we would use the same Department, and Salary example relations used in Table 3.14 and Table 3.15 respectively from section 3.3.2. These relations are reproduced as in Table 3.20 and Table 3.21 to be sorted by Ts, Te, and S.

Table 3.20. Department relation sorted by Ts, Te, & S

Ts	Te	Emp_ID	Dept
1	15	01	1
16	30	01	3
21	60	02	2
31	45	01	1
46	60	01	3
51	60	03	5
61	65	02	3
61	90	03	4
66	69	02	1
70	74	02	2
75	78	02	5
79	80	02	2
90	110	08	2
91	95	04	1
91	140	03	1
96	100	04	2
101	104	04	5
105	110	04	3
111	115	07	2
111	120	04	1
111	130	08	1
116	120	07	1
121	125	07	5
121	126	04	2
126	130	07	1
127	130	04	4
131	134	04	1
131	135	07	5
131	170	08	3
135	138	04	4
136	140	07	3
139	142	04	5
141	150	03	5
141	150	07	4
143	147	04	3
146	150	09	5
148	150	04	1
151	155	09	1
151	160	05	4
151	160	07	1
156	159	09	4
160	168	09	5
161	163	06	1
161	170	07	4
161	180	05	5
164	169	06	2
169	170	09	3
170	174	06	4
171	180	07	2
171	181	09	1
171	190	08	1
175	179	06	3
180	181	06	1
181	190	07	4
181	200	05	3
182	184	06	5
182	193	09	2
185	189	06	3
190	194	06	1
191	200	07	2
191	200	08	2
194	200	09	3
195	197	06	2
198	200	06	5

Table 3.21. Salary relation sorted by Ts, Te, & S

Ts	Te	Emp_ID	Salary
1	10	01	50
11	15	01	55
16	20	01	60
21	30	01	70
21	30	02	60
31	40	01	75
31	40	02	65
41	45	01	80
41	50	02	70
46	50	01	85
51	55	02	75
51	60	01	100
51	60	03	45
56	60	02	85
61	65	02	90
61	65	03	49
66	70	02	100
66	70	03	53
71	80	02	110
71	80	03	58
81	85	03	62
86	90	03	65
90	100	08	40
91	95	03	68
91	95	04	50
96	100	04	55
96	105	03	70
101	105	04	60
101	110	08	45
106	110	03	75
106	110	04	65
111	115	03	80
111	115	04	70
111	115	08	50
111	120	07	50
116	120	04	75
116	125	08	56
116	150	03	85
121	125	04	80
121	130	07	55
126	130	04	85
126	130	08	70
131	135	04	90
131	135	07	60
131	140	08	73
136	140	04	100
136	140	07	65
141	145	04	110
141	150	07	70
141	150	08	75
146	150	04	120
146	150	09	70
151	154	09	85
151	160	05	60
151	160	07	75
151	165	08	78
155	160	09	100
161	170	05	70
161	170	06	60
161	170	09	110
161	175	07	80
166	170	08	80
171	173	08	85
171	180	05	80
171	180	06	70
171	180	09	125
174	175	08	88
176	177	08	90
176	185	07	85
178	181	08	93
181	190	05	90
181	190	06	75
181	190	09	130
182	184	08	95
185	186	08	97
186	195	07	95
187	189	08	99
190	192	08	100
191	200	05	95
191	200	06	80
191	200	09	135
193	193	08	110
194	196	08	115
196	200	07	100
197	200	08	120

Figure 3.10 is a chart of “Department” relation and Figure 3.11 is a chart of “Salary” relation.

Following is the algorithm to implement the join for such temporal relations.

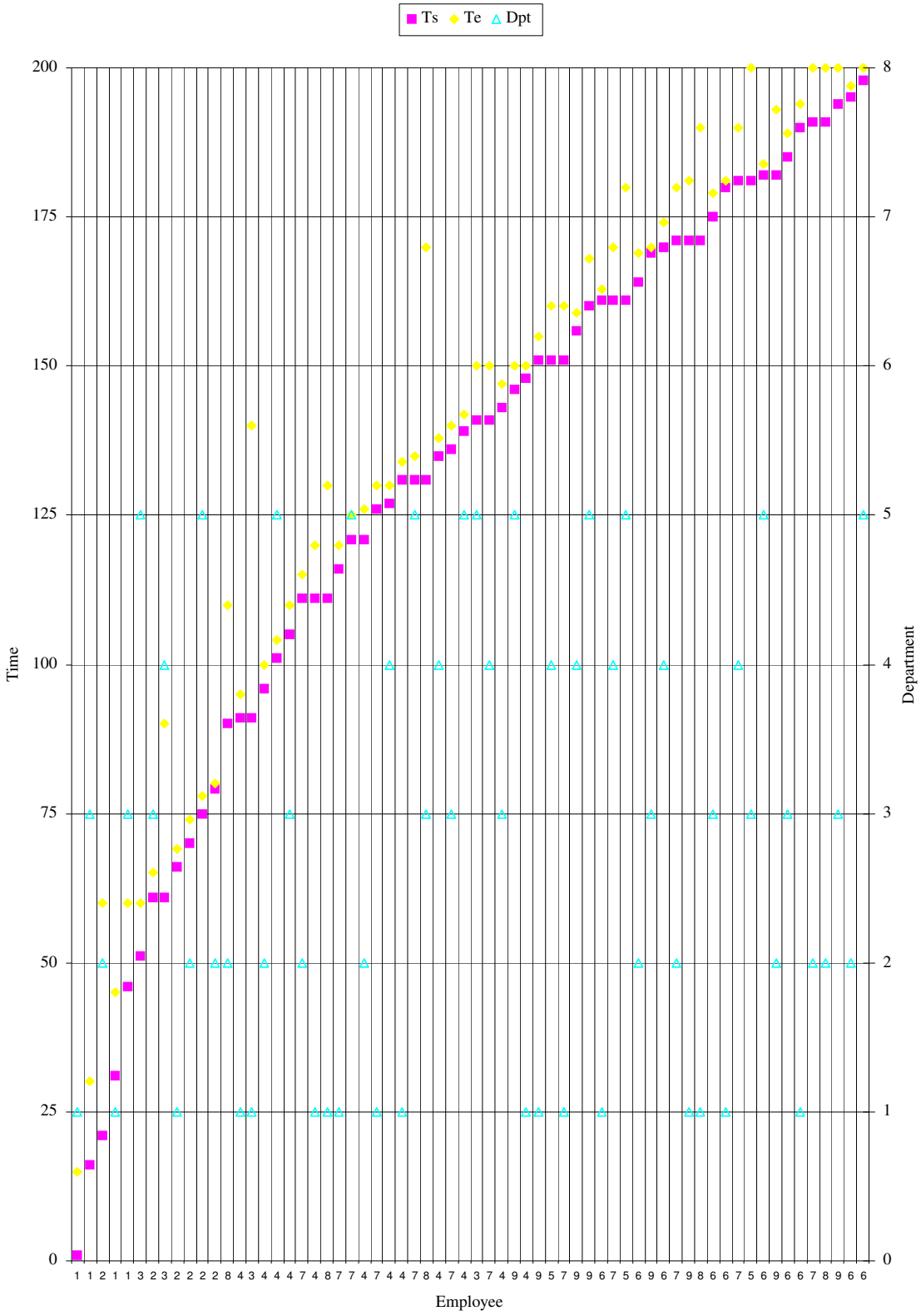


Figure 3.10. Dept relation chart, sorted by Ts, Te & S

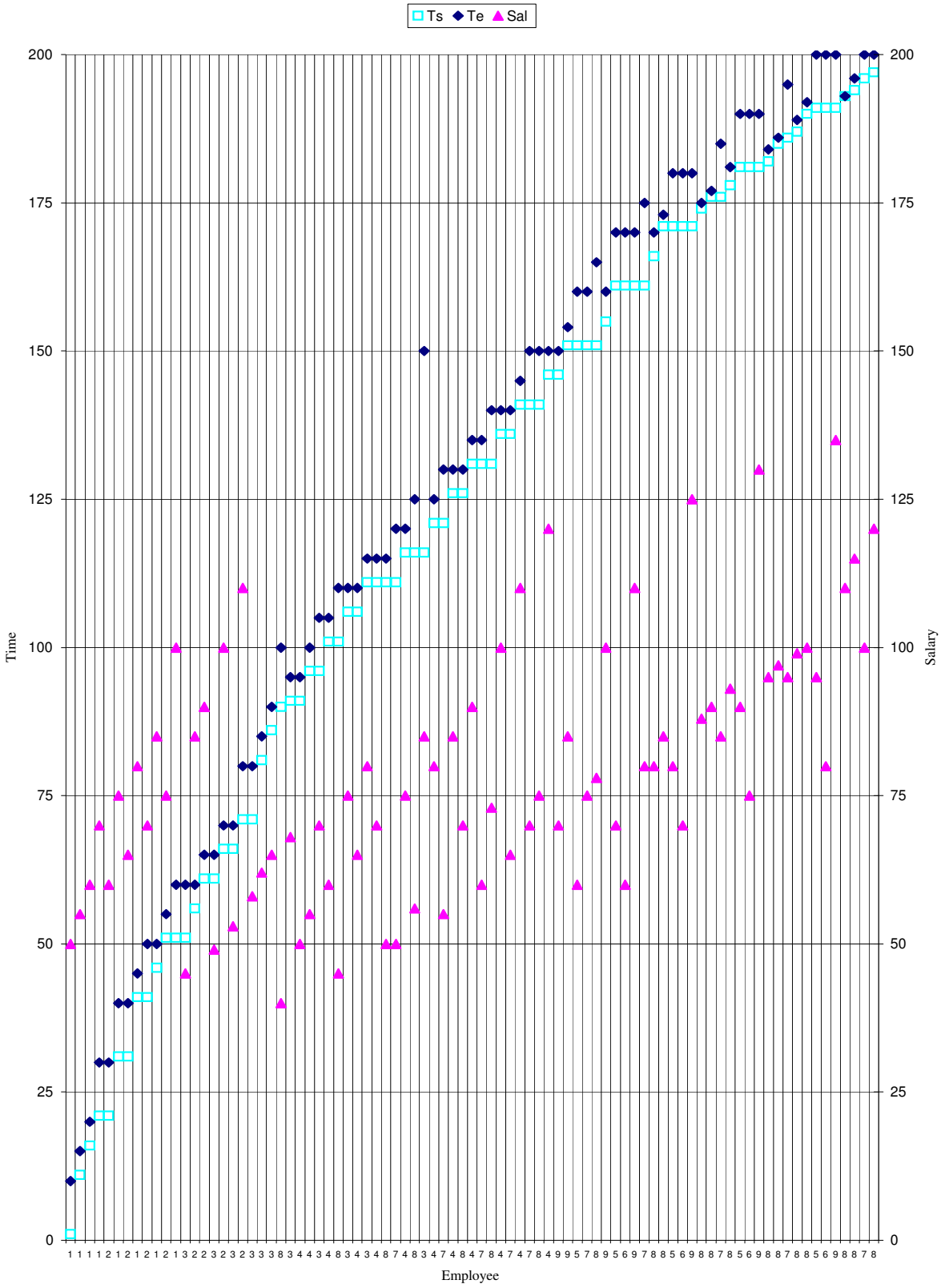


Figure 3.11. Salary relation chart, sorted by Ts, Te & S

Algorithm 3.5. TE-join for relations that are sorted by Ts, Te, and S. Where the link is based on time intersection plus surrogate attribute.

Step 1 : Select next *r1* from R_1 until eof.
Step 2 : Seek *r1* in R_2, if not found step 4
Step 3 : Add tuple to output go to step 1
Step 4 : Find equivalent *Ts* and *S* in R_2, if not found step 7
Step 5 : Produce output, if not $Te_1 > Te_2$ then step 1
Step 6 : Assign uncovered portions in step 5 as *r1*, go step 2
Step 7 : Find r_2 in which $Ts_2 < Ts_1 <= Te_2$ go to step 5

This algorithm is good for event dependent tuples between different relation because they always match in Ts, Te, and S.

Since both relations are indexed on Ts, Te, and S then single loop join would be the most suitable method to implement the join. Either relation can be used as outer loop and the result of the output would be the same in both cases.

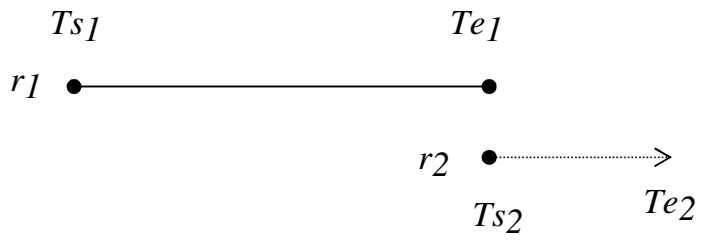
Cost:

In this algorithm, for each record in *R*1 we seek for the corresponding records in *R*2. Hence, *R*1 is scanned only once. In the other side we may find the exact corresponding record in *R*2 directly, or it might be necessary to go back for a tuple in *R*2 were Ts2<Ts1<=Te2. This tuple always exist in the last occurrence of corresponding "S" value from which

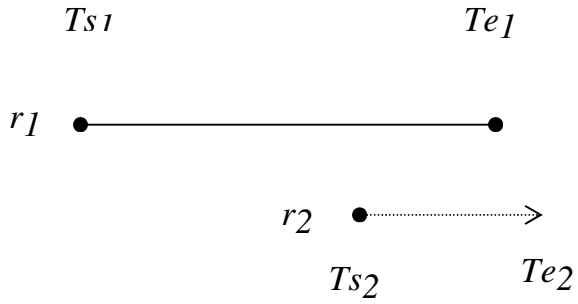
seek pointer start searching using the available index. When the searching pointer stops due to unsuccessful hit.

Cost of this algorithm as follows for each step: In step 1 we start taking tuples from *r*1 in order. In step 2 we seek for matching tuples in *R*2 with regard to primary key which is a multi-value keys consist of Ts, Te, and S. If the matching record is found the cost would be only one block access to produce an output tuple. And next we seek for next record in *R*1. Otherwise, (if not found) we seek for a record that have the same Ts and S using the same index. Again if this tuple were found then most probably the cost would be only one block access to produce a single output tuple (assuming normal distribution).

In step 5 and step 6 if Te1 is longer than Te2 then the uncovered portion of r1 is calculated. And will be searched for again. In step 7, if no such record can be found where Ts1=Ts2 and S1=S2 then we search for a tuple in which Ts2<Ts1<=Te2 as depicted in Figures 3.12 (a) and (b) in order to guarantee intersection



(a)



(b)

Figure 3.12. Tuples r1 and r2 time intersection

In case of step 7, since exact matching Ts and S can not be found, the pointer will stop at r_2 tuple where Ts2 > Ts1. If we search backward from that point to a tuple where we have matching S tuple, the first occurrence of this matching tuple would be our target tuple where Ts2<Ts1<=Te2. The cost to find this tuple would be either in the same block or nearby

block since records are ordered according to Ts time stamp. In step 7 we assume that the pointer stops at r_{21} and our desired tuple is r_{22}. We denote D to be the distance between of r_{21} and r_{22} tuples. Dr represent the number of records occupy this distance. The cost of this algorithm would be:

$$C_{5.3} = b_{r1} + (| r_1 | * (D_{r2} / bfr))$$

If this algorithm were used for relations that have dependent events tuples the cost would drop to be

$$C_{5.3} = b_{r1} + b_{r2}$$

The cost would be as in merge-scan join.

For a better performance of this algorithm, outer relation should be the relation with less number of records in order to increase the number of successful seek (hits) and at the same time reduce the number of unsuccessful seek (hits).

This algorithm has been implemented. The implementation can be found in Annex 1. The output of this implementation is illustrated in Table 3.22. The source code was written in Microsoft Access.

Table 3.22. Output of algorithm 3.5.

Ts	Te	Emp_ID	Dept	Salary
1	10	01	1	50
11	15	01	1	55
16	20	01	3	60
21	30	01	3	70
21	30	02	2	60
31	40	01	1	75
31	40	02	2	65
41	45	01	1	80
41	50	02	2	70
46	50	01	3	85
51	55	02	2	75
51	60	01	3	100
51	60	03	5	45
56	60	02	2	85
61	65	02	3	90
61	65	03	4	49
66	69	02	1	100
66	70	03	4	53
70	70	02	2	100
71	74	02	2	110
71	80	03	4	58
75	78	02	5	110
79	80	02	2	110
81	85	03	4	62
86	90	03	4	65
90	100	08	2	40
91	95	03	1	68
91	95	04	1	50
96	100	04	2	55
96	105	03	1	70
101	104	04	5	60
101	110	08	2	45
105	105	04	3	60
106	110	03	1	75
106	110	04	3	65
111	115	03	1	80
111	115	04	1	70
111	115	07	2	50
111	115	08	1	50
116	120	04	1	75
116	120	07	1	50
116	125	08	1	56
116	140	03	1	85
121	125	04	2	80
121	125	07	5	55
126	126	04	2	85
126	130	07	1	55
126	130	08	1	70
127	130	04	4	85
131	134	04	1	90
131	135	07	5	60
131	140	08	3	73
135	135	04	4	90
136	138	04	4	100
136	140	04	5	100
136	140	07	3	65
141	142	04	5	110
141	150	03	5	85
141	150	07	4	70
141	150	08	3	75
143	145	04	3	110
146	147	04	3	120
146	150	09	5	70
148	150	04	1	120
151	154	09	1	85
151	160	05	4	60
151	160	07	1	75
151	165	08	3	78
155	155	09	1	100
156	159	09	4	100
160	160	09	5	100
161	163	06	1	60
161	168	09	5	110
161	170	05	5	70
161	170	07	4	80
164	169	06	2	60
166	170	08	3	80
169	170	09	3	110
170	170	06	4	60
171	173	08	1	85
171	174	06	4	70
171	175	07	2	80
171	180	05	5	80
171	180	09	1	125
174	175	08	1	88
175	179	06	3	70
176	177	08	1	90
176	180	07	2	85
178	181	08	1	93
180	180	06	1	70
181	181	06	1	75
181	181	09	1	130
181	185	07	4	85
181	190	05	3	90
182	184	06	5	75
182	184	08	1	95
182	190	09	2	130
185	186	08	1	97
185	189	06	3	75
186	190	07	4	95
187	189	08	1	99
190	190	06	1	75
190	190	08	1	100
191	192	08	2	100
191	193	09	2	135
191	194	06	1	80
191	195	07	2	95
191	200	05	3	95
193	193	08	2	110
194	196	08	2	115
194	200	09	3	135
195	197	06	2	80
196	200	07	2	100
197	200	08	2	120
198	200	06	5	80

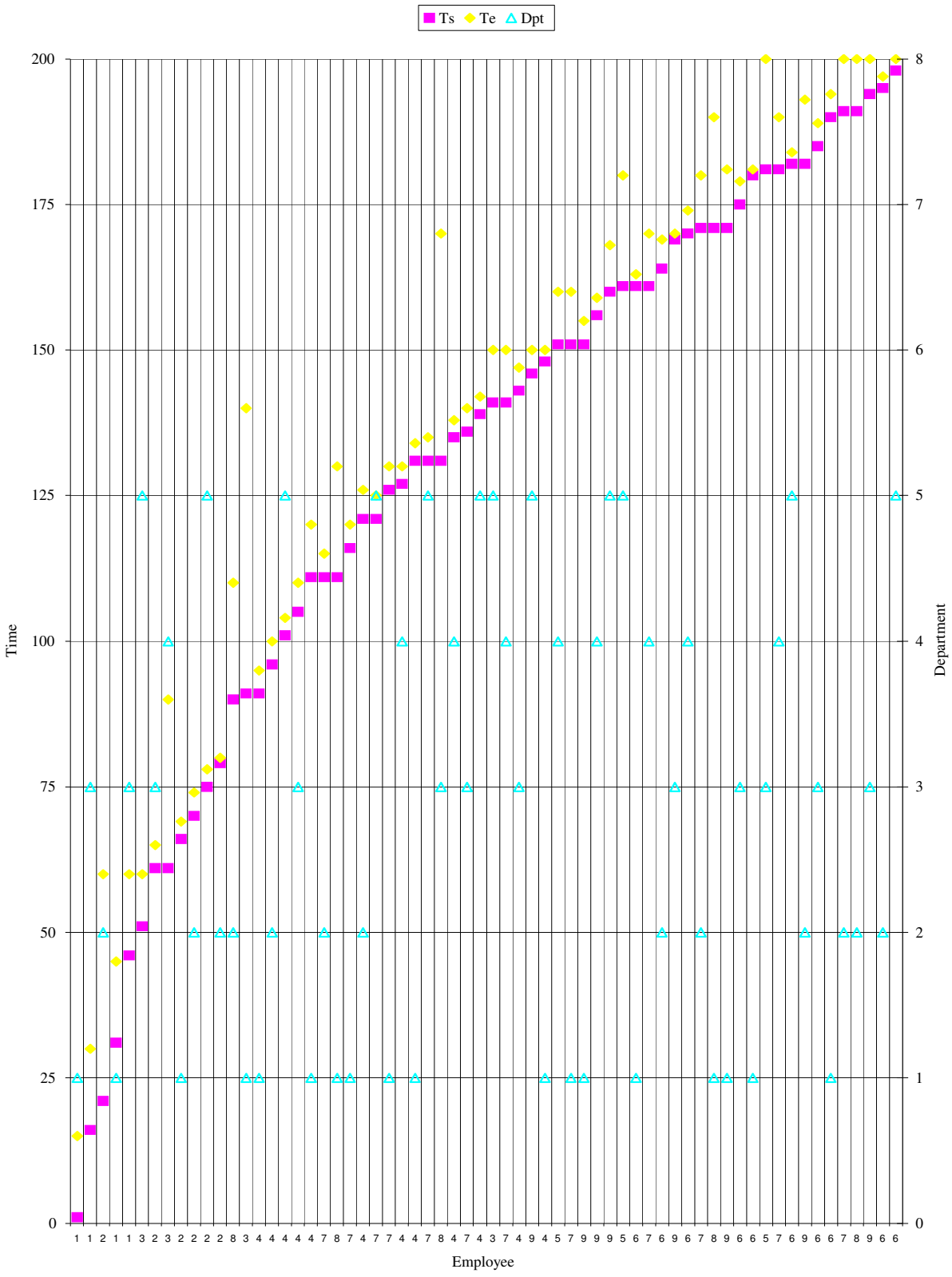
3.3.5 Join Using Ts in a Relation with Itself

For the department relation. Suppose that we need to know employees that have been working together in the same department at the same time. In order to answer this query we have to join "Department" relation with itself. Before we talk about the design of the algorithm to implement this query, let us sort department relation only by "Ts" key. The sorted relation is illustrated in Table 3.23.

Table 3.23. Department relation sorted by Ts

Ts	Emp_ID	Te	Dept
1	01	15	1
16	01	30	3
21	02	60	2
31	01	45	1
46	01	60	3
51	03	60	5
61	02	65	3
61	03	90	4
66	02	69	1
70	02	74	2
75	02	78	5
79	02	80	2
90	08	110	2
91	03	140	1
91	04	95	1
96	04	100	2
101	04	104	5
105	04	110	3
111	04	120	1
111	07	115	2
111	08	130	1
116	07	120	1
121	04	126	2
121	07	125	5
126	07	130	1
127	04	130	4
131	04	134	1
131	07	135	5
131	08	170	3
135	04	138	4
136	07	140	3
139	04	142	5
141	03	150	5
141	07	150	4
143	04	147	3
146	09	150	5
148	04	150	1
151	05	160	4
151	07	160	1
151	09	155	1
156	09	159	4
160	09	168	5
161	05	180	5
161	06	163	1
161	07	170	4
164	06	169	2
169	09	170	3
170	06	174	4
171	07	180	2
171	08	190	1
171	09	181	1
175	06	179	3
180	06	181	1
181	05	200	3
181	07	190	4
182	06	184	5
182	09	193	2
185	06	189	3
190	06	194	1
191	07	200	2
191	08	200	2
194	09	200	3
195	06	197	2
198	06	200	5

The chart of the above relation is illustrated in Figure 3.13. Note from the chart that sorting in "Ts" attribute alone is good enough. Clearly, if there is no second key in the sort, or if there is a second sort key for "Te" or "Dept" attribute, the algorithm cost would still be the same



Figur 3.13 Department relation chart, sorted by Ts

Figure 3.13. Department relation chart, sorted by Ts

Algorithm 3.6 below has been written to find the intersection in time and time attribute inside the same relation. Note that intersection in time and surrogate attribute in the same relation leads to duplicate in temporal tuples, which contradict with temporal normal form.

Algorithm 3.6. TE-join for Same relation that are sorted by Ts. Where the link is based on time intersection plus time attribute.

Step 1 :	If eof then end
Step 2 :	Read $r1$ from R and mark it BOF
Step 3 :	Read next record as r_2
Step 4 :	If not $Ts_2 > Te_1$ then step 6
Step 5 :	- Move pointer to BOF, - Move to next record, - Go to step 1.
Step 6 :	If not $A_1 = A_2$ then step 3
Step 7 :	- Produce output tuple, - Go to step 3.

Single loop join topology is implemented in this algorithm.

Cost:
Note from the chart in Figure 3.13 that each record is probed only once against the n subsequent tuples in the relation. And never probed against any previous tuples, because tuples are sorted in "Ts" attribute in ascending order. in the best case $n = 1$. In the worst case n could be all the subsequent tuples to the end of the relation. Therefore total n in this case would be:

$$C = \sum_{i=1}^{n-1} I$$

$$= n(n+1)/2 - n$$

$$= N^2/2 - n/2$$

So the cost will be: $C_{5.4} = br + (br)^2$

This algorithm has been implemented. The implementation is attached in Annex 1. Microsoft Access was used to write the source code. Table٣,٢٤ illustrates the result of this implementation.

Table 3.24. Output of algorithm 3.6.

Ts	Te	Dept	Emp_ID1	Emp_ID2
96	100	2	08	04
91	95	1	04	03
111	120	1	03	04
111	130	1	03	08
116	120	1	03	07
126	130	1	03	07
131	134	1	03	04
111	120	1	04	08
116	120	1	04	07
116	120	1	08	07
126	130	1	08	07
136	140	3	08	07
143	147	3	08	04
169	170	3	08	09
141	142	5	04	03
146	150	5	03	09
151	155	1	09	07
156	159	4	05	09
161	168	5	09	05
170	170	4	07	06
171	181	1	09	08
180	181	1	09	06
180	181	1	08	06
190	190	1	08	06
185	189	3	05	06
194	200	3	05	09
191	193	2	09	07
191	193	2	09	08
191	200	2	07	08
195	197	2	07	06
195	197	2	08	06

3.3.6 Other Cost Factors

In all the preceding algorithms there are other factors in calculating the cost such like the cost of writing the output (joined tuples). But since the

cost of writing the output tuples to the disk is the same regardless of the chosen algorithm or plan [Silberschatz, 1997]. Then this cost was excluded from our cost analysis for the previous algorithms. For algorithms 3.3, 3.4, and 3.5 above, this cost is estimate to be:

$$C_{\text{write-to-disk}} = (J_S * | R_1 | * | R_2 |) / bfr$$

For algorithms 3.6 above, This cost is estimate to be:

$$C_{\text{write-to-disk}} = (J_S * | R_1 | * | R_1 |) / bfr$$

Where *Js* is the join selectivity.

The other type of cost that might be associated with the above algorithms is sorting cost, in case the joined relation is not sorted. As the disk access is the major factor in calculating the cost, then external sorting cost have to be calculated. The most famous external sorting techniques is external sort-merge algorithm. The cost of using this algorithm is :

$$C_{\text{external sort-merge}} = b_r (2 \lceil \log_{M-1} (b_r/M) \rceil + 1)$$

Where *M* is the number of buffer pages.

The sorting cost can be calculated separately and added to the total cost of the algorithm if necessary.

4. CONCLUSION

4.1 Summary

In this dissertation, three main areas have been discussed. These areas are involved in query optimization in relational databases in general and in temporal relational databases in particular. These areas are relational database schemes, relational database indexes, and temporal relations joining in temporal relational databases.

We have suggested in section 3.1 a schema for temporal relational databases. Tests have shown that tuple time stamping that involves multiple relations for every time related attribute has a better overall performance and efficiency, in both, processing time and used space. Time representations in temporal database have a great effect in query optimization. Time stamps should not occupy more memory space if they can be managed to occupy less memory space, not only to save memory, but also to improve the processing time.

Some researchers have concentrated on extending regular relational concepts to adapt temporal aspects. Others have came up with totally new ideas that even violate basic rules of relational databases. A reasonable

combination of both should be taken in consideration when we want to create a reliable and efficient temporal database. Integrating the time dimension with relational model efficiently have to be accompanied with a remedies to the complexities that arises from such a process. These obstacles are listed in section one.

Abstract interfacing modules that regulate working with temporal information have to be used in temporal databases. Also, the database catalogues (dictionary) have to accommodate the temporal aspect with regard to temporal relation.

Time behavior is predictable. It can be anticipated accurately. These characteristics have been explored in creating a new approach that uses hash function in indexing temporal relational databases. The cost of this index is always one disk access.

There are applications were accurate timing for valid time-stamps are not so crucial. As an example, application where data is updated upon investigation. At the time of investigation, the status of the time attribute might be found to be altered. But the exact alteration time is unknown. In these kind of application transaction time can represent valid time too.

Continuity or discontinuities in temporal events in relational database have to be taken into account when designing algorithms to join temporal relation together. In section 3.3 we have explored four different types of

algorithms. Each one of these algorithms is dedicated to deal most efficiently with a temporal relation in a specific sorting order. The disk access cost for these algorithms was discussed too. In order to ensure the correctness of these four algorithms, these algorithms have been implemented.

4.2 Future Work

In section 3.1, user-defined time-stamps have been suggested. Less time-stamps size will lead to less numbers of disk access. But investigation needs to be made with regards to the overhead in processing time caused by processing varies sizes of time-stamps. Does the saved time in disk access overweight the overhead caused by processing various sizes of time-stamps?

Investigate the possibility and the implications of using a sequential real number instead of the suggested used time granularity in the hash clustered index structure approach.

Testing of the suggested schema can be implemented in real life application with build-in constructs using universal time time-stamping. Implementation of testing will investigate the feasibility and efficiency of the suggested schema. Comparison of the suggested hashed cluster index with other available schemes can be made to evaluate its performance.

Although we have explored only four algorithms in joining temporal relations of continuous type time event nature, but certainly there are more algorithms to be explored for temporal relations of discrete type time event nature. Also joins that involves time attribute in one relation with surrogate attribute in the other relation with different sorting order have not been investigated yet.

A complete algorithm to optimize a query in regular relation databases have been explored as reviewed in section 2. In temporal relational database, it is still early to develop such complete optimization algorithms. But we can at least study the space search algorithms to evaluate the cost in temporal relational databases for queries where temporal attribute and none-temporal attributes are involved together in the same query. Semijoins have been disscussed by Chen [1990]. We may explore the possibilities to utilize semijoins with joining non-temporal relations with temporal relations.

It has been expected that the next twenty years will be as active as the previous twenty and will bring many advances to query optimization technology. To describe the continuous importance of query optimization in databases, we close the thesis by a quotation from Ioannidis [1995] describing the research in this area, he wrote, “Despite its age, query optimization remains an exciting field of researches”.

References:

Ahn, I., and Snodgrass, R.. 1986. Performance Evaluation of a Temporal Database Management System. ACM Publication 0-89791-191-1/86/0500/0096, pp 96-107.

Aho, V., Sagiv, Y., and Ullman, J.. 1986. *Compilers: Principles, Techniques, and Tools*. Addison Wesley.

Bair, J., Bohlen, M., Jensen, C., and Snodgrass, R.. 1997. Notions of Upward Compatibility of Temporal Query Languages. Prepared for CHOROCHRONOS Project Funded by European commission. pp 589-613.

Bozkaya, T.. 1998. Index Structures for Temporal and Multimedia Databases. For the Degree of Doctor of Philosophy. Case Western Reserve University.

Chen, M., and Yu, P.. 1990. Interleaving A Join Sequence With Semijoins in Distributed Query Processing. IBM Thomas J. Watson Research Center, Yorktown Heights, New York.

Chomicki, J.. 1995. Temporal Query Languages: a Survey. ACM Symposium on Principles of Database systems

Christodoulakis, S.. 1984. Implications of certain assumptions in database performance evaluation. ACM TODS, 9(2) pp 163-186.

Delaney, C., Rama, D., and Srinivasan, P.. 1992. Design of a Temporal Database for Phlebitis. ACM 0-89791-502-X/92/0002/0204, pp 204-209.

Dyreson, C., and Snodgrass, R.. 1994. Temporal Granularity in TSQL2. University of Arizona Technical Report 94-06. Arizona, United States of America.

Elmasri, R., Kouramajian, V., and Fernando, S.. 1993. Temporal Database Modeling: An Object-Oriented Approach. ACM 0-89791-626-3/93/0011, pp 574-585.

Elmasri, R., Navathe, S.. 2000. *Fundamentals of Database Systems*. 3rd edition. Addison Wesley. Canada.

Gadia, S., and Yeung, C.. 1988. A Generalized Model for a Relational Temporal Database. ACM Publication 0-89791-3/88/0006, pp 251-259.

Gadia, S.. 1988. A Homogeneous Relational Model and Query Languages for Temporal Databases. ACM Trans on Databases Systems, Vol. 3, No. 4, pp 418-448.

Goralwalla, I., Tansel, A., and Ozsu, M.. 1995. Experimenting with Temporal Relational databases. ACM Publication 0-89791-812-6/95/11, pp 596-303.

Gunadhi, H., and Seqev, A.. 1990. A Framework in Temporal Databases. Springer Verlag, Volume 420, p 131-147.

Ibaraki, T., and Kameda, T.. 1984. On the Optimal Nesting Order for Computing N-Relation Joins. ACM Trans. Database Syst. 9(3), pp 482-502

Ioannidis, Y., and Wong, E.. 1987. Query Optimization by Simulated Annealing. Proc. ACM SIGMOD Int. Connf. On Management of Data, pp 9-22

Ioannidis, Y., and Kang, Y.. 1990. Randomized Algorithms for Optimizing Large Join Queries. Proc. ACM-SIGMOD Conference of the Management of Data, pp 312- 321.

Ioannidis, Y., and Kang, Y.. 1991. Left-Deep vs. Bushy Trees: An Analysis of Strategy Spaces and Its Implications for Query Optimization. SIGMOD.

Ioannidis, Y. and Christodoulakis, S.. 1993. Optimal Histograms for Limiting Worst-case Error Propagation in the Size of Join Results. ACM TODS, 18(4), pp 709-748.

Ioannidis, Y.. 1995. Query Optimization. University of Wisconsin,

Wisconsin, United States of America.

Kim, W.. 1982. On Optimizing an SQL-like Nested Query. TODS, 3:3.

King, J.. 1981. QUIST: A System for Semantic Query Optimization in Relational Databases. VLDB Conference.

Knuth, D.. 1973. *The Art of Computer Programming*. Vol 3, Addison Wesley, Sorting and Searching.

Kooi, R.. 1980. The optimization of Queries in Relational Databases. PhD thesis, Case Western Reserve University.

Kouramajian, V., Kamal, I., Elmasri, R., and Waheed, S.. 1994. The Time Index^{+}: An Incremental Access Structure for Temporal Databases. ACM Publication 0-89791-654-3/94/0011, pp 296-303.

Lipton, R., Naughton, J., and Schneider, D.. 1990. Practical Selectivity Estimation through Adaptive Sampling. SIGMOD.

Nascimento, M., and Eich, M.. 1995. Indexing Bitemporal Databases Via Trees with Shared Leaves – The SLT Approach. Technical Report 95-CSE-06, Southern Methodist University. Texas, United States of America

Ozkaraham, E.. 1990. *Database Management Concepts, Design, and Practice*. Prentice Hall International, Inc. New Jersey. United States of America.

Ozsu, M., and Valduriez, P.. 1999. *Principles of Distributed Database Systems*. 2nd edition. Prentice-Hall, Inc. New Jersey, United States of America.

Piatetsky-Shapiro, G., and Connell, C.. 1984. Accurate Estimation of the Number of Tuples Satisfying a Condition. Proc ACM-SIGMOD Conference of the Management of Data, PP 256-276.

Popescul, A., Gary F., Steve L., Lyle U., and Giles, C.. 2000. Clustering and Identifying Temporal Trends in Document Database. IEEE Advances in Digital Libraries, pp 173-182.

Qutaishat, M.. 1999. *Databases*. The Arab Academy for Banking and Financial Sciences. Jordan.

Segev, A. and Shoshani, A.. 1998. Functionality of Temporal Data Models and Physical Design Implementations. IEEE Database Engineering, 11(4):38-45.

Selinger, P., Astrahan, M., Chamberlin, D., Lorie, R., and Price, T.. 1979. Access Path Selection in a Relational Database Management System. Proc. ACM-SIGMOD Conf. On the Management of Data, PP 23-34.

Siegel, M., Sciore, E., and Salveter, S.. 1992. A Method for Automatic Rule Derivation to Support Semantic Query Optimization. TODS, 17:4.

Silberschatz, A., Korth, H., and Sudarshan, S.. 1997. *Database Systems Concepts*. 3rd edition. The McGraw Hill Companies, Inc. Singapore.

Skjellaug, B.. 1997. Temporal Data: Time and Object databases. Universitetet i Oslo, Norway

Slivinskas, G., Jensen, C., and Snodgrass, R.. 2001. Adaptable Query Optimization and Evaluation in Temporal Middleware. A TimeCenter Technical Report.

Smith, J., and Chang, P.. 1975. Optimizing the Performance of a Relational Algebra Database Interface. Commun. ACM, 18(10): pp 568-579

Snodgrass, R.. 1987. The Temporal Query Language TQuel. ACM Trans on Database Systems, Vol. 12, No. 2, pp 247-298.

Snodgrass, R., Bohen, M., Jensen, C., and Steiner, A.. 1996. Adding Transaction Time to SQL / Temporal. International Organization for Standardization. ANSI X3H2-96-502r2.

Snodgrass, R., Bohen, M., Jensen, C., and Steiner, A.. 1996. Adding Valid

Time to SQL / Temporal. International Organization for Standardization. ANSI X3H2-96-501r2.

Spiteri, M., and Bates, J.. 1998. An Architecture to Support Storage and Retrieval of Events. Proceeding of MIDDLEWARE, IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing

Steiner, A.. 1998. A Generalization Approach to Temporal Data Models and their Implementations. A dissertation submitted for the degree of Doctor of Technical sciences. Swiss Federal Institute of Technology. Zurich, Switzerland.

Swami, A.. 1989. Optimization of Large Join Queries: Combining Heuristics and Combinatorial Techniques. Proc ACM SIGMOD Int. Conf. On Management of Data,, pp 367-376

Tansel, A., Clifford, J., Gadia, S., Jajodia, S., Segev, A., and Snodgrass, R.. 1993. *Temporal Databases Theory, Design, and Implementation*. The Benjamin / Cummings Publishing Company, Inc. California. United States of America.

Toman, D.. 1995. Foundations of Temporal Query Languages. A Dissertation for Doctor of Philosophy. Kansas State University. Kansas, United States of America.

Toman, D.. 1996. Point vs. Interval-based Query Languages for Temporal Databases. ACM 0-89791-791- 2/96/06. pp 58-67

Toshiyuki A., Masayoshi A., and Yoshinari K. 1999. An Implementation of Interval Based Conceptual Model for Temporal Data. IEICE Trans. Inf. & Syst., Vol. E82-D, No. 1, pp 136-146.

Tsotras, V., and Kumar, A.. 1996. Temporal Database Bibliography Update. SIGMOD Record, 25(1): pp 41-51.

Ullman, J.. 1982. *Principles of Database Systems* (2nd edition), Md.: Computer Science Press.

Valduriez, P., and Gardarin, G.. 1984. Join and Semi-join Algorithms for a Multi Processor Database Machine. ACM Trans. Database Syst. 9(1), pp 133-161.

Yoo, H. and Lafortune, S.. 1989. An Intelligent Search Method for Query Optimization by Semijoins. IEEE Trans. On Knowledge and Data Engineering 1(2): pp 226-237.

Yu, X., and Fu, A.. 1984. Piecewise Linear Histograms for Selectivity Estimation. SIGMOD, ACM Press, pp 256-276.

Appendix 1 : Join implementations source code

Module: Join_Dpt&Dpt_Ts&A Page: 1

Properties

Date Created:	09/03/2002 2:16:29 PM	Last Updated:	10/03/2002 8:28:46 AM
Owner:	admin		

Code

```
Attribute VB_Name = "Join_Dpt&Dpt_Ts&A"
Option Compare Binary
Option Explicit



Public Function Join4()

  Dim MyDb As Database
  Dim Dpt, Join As Recordset
  Dim DTs1, DTe1, DTs2, DTe2 As Byte
  Dim Dpt1, Dpt2 As String
  Dim Emp1, Emp2 As String
  Dim VarBook1 As Variant
  Dim RecNo As Double

  Set MyDb = CurrentDb()
  Set Dpt = MyDb.OpenRecordset("Emp_Temp_Dept", DB_OPEN_TABLE)
  Set Join = MyDb.OpenRecordset("Emp_Dpt&Dpt_Ts+A", DB_OPEN_TABLE)

  Dpt.Index = "TSATE"
  Dpt.MoveLast
  Dpt.MoveFirst

  DTs1 = Dpt![TS]
  DTe1 = Dpt![TE]
  Dpt1 = Dpt![Dept]
  Emp1 = Dpt![Emp_ID]
  VarBook1 = Dpt.Bookmark
  RecNo = 0

Start:
'================================================================
  Dpt.MoveNext
   If Not Dpt.EOF Then
      DTs2 = Dpt![TS]
      DTe2 = Dpt![TE]
      Dpt2 = Dpt![Dept]
      Emp2 = Dpt![Emp_ID]
   End If

```

```
42          If (DTs2 > DTe1 Or Dpt.EOF) Then
43             Dpt.Bookmark = VarBook1
44             Dpt.MoveNext
45             If Dpt.EOF Then
46               MsgBox "Joint process is completed"
47               Exit Function
48             End If
49             VarBook1 = Dpt.Bookmark
50             DTs1 = Dpt![TS]
51             DTe1 = Dpt![TE]
52             Dpt1 = Dpt![Dept]
53             Emp1 = Dpt![Emp_ID]
54             GoTo Start
55         Else
56             If Dpt1 <> Dpt2 Then
57                GoTo Start
58             Else
59                Select Case DTe1
60                Case Is > DTe2 ''' DTe1 > DTe2
61                   RecNo = RecNo + 1
62                   Join.AddNew
63                      Join![Emp_ID1] = Emp1
64                      Join![Emp_ID2] = Emp2
65                      Join![Dept] = Dpt1
66                      Join![TS] = DTs2
67                      Join![TE] = DTe2
68                      Join![No] = RecNo
69                   Join.Update
70                   GoTo Start
71                Case Is = DTe2 ''' DTe1 = DTe2
72                   RecNo = RecNo + 1
73                   Join.AddNew
74                      Join![Emp_ID1] = Emp1
75                      Join![Emp_ID2] = Emp2
76                      Join![Dept] = Dpt1
77                      Join![TS] = DTs2
78                      Join![TE] = DTe2
79                      Join![No] = RecNo
80                   Join.Update
81                   GoTo Start
82                Case Else      ''' DTe1 < DTe2
83                   RecNo = RecNo + 1
84                   Join.AddNew
85                      Join![Emp_ID1] = Emp1
86                      Join![Emp_ID2] = Emp2
87                      Join![Dept] = Dpt1
88                      Join![TS] = DTs2
89                      Join![TE] = DTe1
90                      Join![No] = RecNo
91                   Join.Update
92                   GoTo Start
```

```
93                  End Select
94
95              End If
96
97          End If
98
99      End Function
```

User Permissions

admin

Group Permissions

Admins
Users

Properties

Date Created:	05/03/2002 11:33:49 AM	Last Updated:	06/03/2002 2:07:42 PM
Owner:	admin		

Code

```
1       Attribute VB_Name = "Join_Sal&Dept_S&Ts"
2       Option Compare Database
3       Option Explicit
4
5
6       Public Sub Join1()
7         Dim MyDb As Database
8         Dim MySetDept, MySetSal, MySetJoin As Recordset
9         Dim Dpt, Sal, Join As Recordset
10        Dim i, j As Double
11        Dim DTs, DTe, STs, STe As Byte
12
13        Set MyDb = CurrentDb()
14        Set Sal = MyDb.OpenRecordset("Emp_Temp_Sal", DB_OPEN_TABLE)
15        Set Dpt = MyDb.OpenRecordset("Emp_Temp_Dept", DB_OPEN_TABLE)
16        Set Join = MyDb.OpenRecordset("Emp_Sal&Dept_S&Ts", DB_OPEN_TABLE)
17
18        Dpt.Index = "primarykey"
19        Sal.Index = "primarykey"
20        Dpt.MoveFirst
21        Sal.MoveFirst
22
23        DTs = Dpt![TS]
24        DTe = Dpt![TE]
25        STs = Sal![TS]
26        STe = Sal![TE]
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
```

```
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57        Start:
58        '================================================================
59        If Dpt![Emp_ID] = Sal![Emp_ID] Then
60
61            If DTs = STs Then
62
63                  Select Case DTe
64                    Case Is > STe ' DTe > STe
65                      Join.AddNew
66                        Join![Emp_ID] = Dpt![Emp_ID]
67                        Join![Dept] = Dpt![Dept]
68                        Join![Salary] = Sal![Salary]
69                        Join![TS] = DTs
70                        Join![TE] = STe
71                      Join.Update
72                      DTs = STe + 1
73                      DTe = DTe
74                      Sal.MoveNext
75                      STs = Sal![TS]
76                      STe = Sal![TE]
77                      GoTo Start
78                    Case Is = STe ' DTe = STe
79                      Join.AddNew
80                        Join![Emp_ID] = Dpt![Emp_ID]
81                        Join![Dept] = Dpt![Dept]
82                        Join![Salary] = Sal![Salary]
83                        Join![TS] = DTs
84                        Join![TE] = DTe
85                      Join.Update
86                      Dpt.MoveNext
87                      Sal.MoveNext
88                      DTs = Dpt![TS]
89                      DTe = Dpt![TE]
90                      STs = Sal![TS]
91                      STe = Sal![TE]
92                      GoTo Start
```

```
 93                    Case Else  ' DTe < STe
 94                      Join.AddNew
 95                        Join![Emp_ID] = Dpt![Emp_ID]
 96                        Join![Dept] = Dpt![Dept]
 97                        Join![Salary] = Sal![Salary]
 98                        Join![TS] = DTs
 99                        Join![TE] = DTe
100                      Join.Update
101                      STs = DTe + 1
102                      STe = STe
103                      Dpt.MoveNext
104                      DTs = Dpt![TS]
105                      DTe = Dpt![TE]
106                      GoTo Start
107                  End Select
108
109             End If
110         else
111         Msgbox "Alert..Invistigate"
112         End If
113
114         End Sub
```

User Permissions

admin

Group Permissions

Admins
Users

Module: Join_Sal&Dept_Ts&S Page: 7

Properties

Date Created: 09/03/2002 11:46:50 AM Last Updated: 19/03/2002 7:56:52 PM
Owner: admin

Code

```
Attribute VB_Name = "Join_Sal&Dept_Ts&S"
Option Compare Binary
Option Explicit



Public Function Join2()

  Dim MyDb As Database
  Dim Dpt, Sal, Join As Recordset
  Dim DTs, DTe, STs, STe As Byte

  Set MyDb = CurrentDb()
  Set Sal = MyDb.OpenRecordset("Emp_Temp_Sal", DB_OPEN_TABLE)
  Set Dpt = MyDb.OpenRecordset("Emp_Temp_Dept", DB_OPEN_TABLE)
  Set Join = MyDb.OpenRecordset("Emp_Sal&Dept_Ts&S", DB_OPEN_TABLE)

  Dpt.Index = "primarykey"
  Sal.Index = "primarykey"
  Dpt.MoveFirst
  Sal.MoveFirst

  DTs = Dpt![TS]
  DTe = Dpt![TE]

Start1:
'================================================================
  If Dpt.EOF Then
    MsgBox "Joint process is completed"
    Exit Function
  End If

  Sal.Seek "=", DTs, Dpt!Emp_ID
  If Sal.NoMatch Then
        Sal.Seek ">", Dpt!TS     ''', Dpt!Emp_ID
            If Sal.NoMatch Then  ''' to handle eof
                Sal.MoveLast
                Do While Dpt!Emp_ID <> Sal!Emp_ID
                  Sal.MovePrevious
                Loop
            End If
```

```
42
43              STs = Sal![TS]
44              Do While (Dpt!Emp_ID <> Sal!Emp_ID Or STs > DTs)
45                Sal.MovePrevious
46                STs = Sal![TS]
47              Loop
48          End If
49          STs = Sal![TS]
50          STe = Sal![TE]
51
52        '=============================================================
53                  Select Case DTe
54                    Case Is > STe ''' DTe > STe
55                          Join.AddNew
56                            Join![Emp_ID] = Dpt![Emp_ID]
57                            Join![Dept] = Dpt![Dept]
58                            Join![Salary] = Sal![Salary]
59                            Join![TS] = DTs
60                            Join![TE] = STe
61                          Join.Update
62                          DTs = STe + 1
63                          DTe = DTe
64                          GoTo Start1
65                    Case Is = STe ''' DTe = STe
66                          Join.AddNew
67                            Join![Emp_ID] = Dpt![Emp_ID]
68                            Join![Dept] = Dpt![Dept]
69                            Join![Salary] = Sal![Salary]
70                            Join![TS] = DTs
71                            Join![TE] = DTe
72                          Join.Update
73                          Dpt.MoveNext
74                          If Not Dpt.EOF Then
75                            DTs = Dpt!TS
76                            DTe = Dpt!TE
77                          End If
78                          GoTo Start1
79                    Case Else    ''' DTe < STe
80                          Join.AddNew
81                            Join![Emp_ID] = Dpt![Emp_ID]
82                            Join![Dept] = Dpt![Dept]
83                            Join![Salary] = Sal![Salary]
84                            Join![TS] = DTs
85                            Join![TE] = DTe
86                          Join.Update
87                          Dpt.MoveNext
88                          If Not Dpt.EOF Then
89                            DTs = Dpt!TS
90                            DTe = Dpt!TE
91                          End If
92                          GoTo Start1
```

```
93                  End Select
94
95        End Function
```

User Permissions

admin

Group Permissions

Admins
Users

Module: Join_Sal&Dept_Ts&Te Page: 10

Properties

Date Created:	07/03/2002 5:09:49 PM	Last Updated:	08/03/2002 6:51:29 PM
Owner:	admin		

Code

```
 1        Attribute VB_Name = "Join_Sal&Dept_Ts&Te"
 2        Option Compare Database
 3        Option Explicit
 4
 5
 6        Public Function Join3()
 7
 8          Dim MyDb As Database
 9          Dim Dpt, Sal, Join As Recordset
10          Dim DTs, DTe, STs, STe, Tmp1, Tmp2 As Byte
11
12          Set MyDb = CurrentDb()
13          Set Sal = MyDb.OpenRecordset("Emp_Temp_Sal", DB_OPEN_TABLE)
14          Set Dpt = MyDb.OpenRecordset("Emp_Temp_Dept", DB_OPEN_TABLE)
15          Set Join = MyDb.OpenRecordset("Emp_Sal&Dept_Ts&Te", DB_OPEN_TABLE)
16
17          Dpt.Index = "TSTE"
18          Sal.Index = "TSTE2"
19          Dpt.MoveLast
20          Sal.MoveLast
21          Dpt.MoveFirst
22          Sal.MoveFirst
23
24          STs = Sal![TS]
25          STe = Sal![TE]
26          DTs = Dpt![TS]
27          DTe = Dpt![TE]
28
29
30
31        Start1:
32        '================================================================
33          If Dpt.EOF Then
34            MsgBox "Joint process is completed"
35            Exit Function
36          End If
37
38          Sal.Seek "=", DTs, DTe, Dpt!Emp_ID
39          If Sal.NoMatch Then
40              Sal.Seek ">=", DTs ', We use >= to represent = here
41              If Not Sal.NoMatch Then ' neither > nor =, to handle eof
```

```
42              If Sal![TS] <> DTs Then ' STs > DTs, we don't need more seeks
43                    Sal.MovePrevious
44                    Do While Dpt!Emp_ID <> Sal!Emp_ID
45                      Sal.MovePrevious
46                    Loop
47                    If Not (Dpt!TS > Sal!TS And Dpt!TS <= Sal!TE) Then
48                        MsgBox " Alert, Invistigate"
49                    End If
50              Else ' STs = DTs
51                  Do While (Dpt!Emp_ID <> Sal!Emp_ID And DTs = Sal!TS)
52                      Sal.MoveNext
53                      If Sal.EOF Then
54                         Sal.MovePrevious
55                         Exit Do
56                      End If
57                  Loop
58                  If (DTs <> Sal!TS Or Dpt!Emp_ID <> Sal!Emp_ID) Then
59                      Sal.MovePrevious
60                      Do While Dpt!Emp_ID <> Sal!Emp_ID
61                        Sal.MovePrevious
62                      Loop
63                      If Not (Dpt!TS > Sal!TS And Dpt!TS <= Sal!TE) Then
64                          MsgBox " Alert, Invistigate"
65                      End If
66                  End If
67              End If
68              Else
69                  Sal.MoveLast
70                  Do While Dpt!Emp_ID <> Sal!Emp_ID
71                      Sal.MovePrevious
72                  Loop
73                  If Not (Dpt!TS > Sal!TS And Dpt!TS <= Sal!TE) Then
74                      MsgBox " Alert, Invistigate"
75                  End If
76              End If
77          End If
78          STs = Sal![TS]
79          STe = Sal![TE]
80
81        Start2:
82        '================================================================
83                  Select Case DTe
84                    Case Is > STe ' DTe > STe
85                      Join.AddNew
86                        Join![Emp_ID] = Dpt![Emp_ID]
87                        Join![Dept] = Dpt![Dept]
88                        Join![Salary] = Sal![Salary]
89                        Join![TS] = DTs
90                        Join![TE] = STe
91                      Join.Update
92                      DTs = STe + 1
```

```
 93                      DTe = DTe
 94                      GoTo Start1
 95                    Case Is = STe ' DTe = STe
 96                      Join.AddNew
 97                        Join![Emp_ID] = Dpt![Emp_ID]
 98                        Join![Dept] = Dpt![Dept]
 99                        Join![Salary] = Sal![Salary]
100                        Join![TS] = DTs
101                        Join![TE] = DTe
102                      Join.Update
103                      Dpt.MoveNext
104                      If Not Dpt.EOF Then
105                        DTs = Dpt![TS]
106                        DTe = Dpt![TE]
107                      End If
108                      GoTo Start1
109                    Case Else  ' DTe < STe
110                      Join.AddNew
111                        Join![Emp_ID] = Dpt![Emp_ID]
112                        Join![Dept] = Dpt![Dept]
113                        Join![Salary] = Sal![Salary]
114                        Join![TS] = DTs
115                        Join![TE] = DTe
116                      Join.Update
117                      Dpt.MoveNext
118                      DTs = Dpt!TS
119                      DTe = Dpt!TE
120                      GoTo Start1
121                  End Select
122
123       End Function
```

User Permissions

admin

Group Permissions

Admins
Users

Appendix 2 : Abbreviations and description

DBMS	Database Management System
CPU	Central Processing Unit
WAN	Wide Area Network
MAN	Medium Area Network
1NF	First Normal Form
N1NF	Non-First Normal Form
TTSR	Tuple Timestamp Single Relation
TTMR	Tuple Timestamp Multiple Relations
K	Key attributes
U	Non-Temporal attributes (Unchangeable)
M	Temporal attributes
L	Timestamps, L=[l_1,l_2]
S_i	Size of fields in bytes
C(t)	Cost of a tuple in total number of bytes.
P	Probability for a temporal attribute to be updated (changed)
Ts	Timestamp (starting)
Te	Timestamp (ending)
S	Surrogate attribute (Key field in temporal relation)
A	Temporal attribute
TBT	Temporal Buffer Table
UC	Until Changed
TIM	Temporal Interfacing Model
D	Address of most outer index level
N	Number of levels
P	Address pointer
UT	Universal Timing
TNF	Temporal Normal Form
BCNF	Boyce-Codd Normal Form
R	Relation
r	Tuple within a relation
.eof	End Of File
b	Number of blocks
Dr	Number of tuples within a distance between pointer stops and the actual correct address
Seek	Used to search the index
Find	Used to search a sequential file
Bfr	Buffer Factor
Js	Join Selectivity

الاسترجاع الأمثل والأكثر كفاءة في بيئة قاعدة البيانات العلائقية الزمنية

إعداد
سمير عادل محمد

إشراف
الدكتور منيب قطيشات

ملخص

تعتبر قواعد البيانات الزمنية مستودع للمعلومات التي تعتمد علي الزمن. الفرق الرئيسي بينها وبين نظم قواعد البيانات العلائقية هو إمكانية الحاجة لتخزين عدد غير محدود من السجلات التي تزيد مع مرور الزمن.

قدمت اقتراحات عديدة لبناء نماذج تضيف العامل الزمني لنظم قواعد البيانات العلائقية المعيارية. بعض هذه النماذج تقترح استعمال طوابع زمنية للحقول. بينما تقترح النماذج الأخرى استعمال طوابع زمنية أحادية الجدول أو طوابع زمنية متعددة الجداول.

هذه الأطروحة تحاول طرح عدة قضايا متعلقة بنموذج قواعد البيانات العلائقية الزمنية ذي الطوابع الزمنية المتعددة الجداول و تطوير نظم قواعد البيانات العلائقية الزمنية: تم مناقشة نماذج نظم قواعد البيانات العلائقية الزمنية المتعددة. تحديد متطلبات التخزين لنماذج نظم قواعد البيانات العلائقية الزمنية المتعددة الجداول و نظم قواعد البيانات العلائقية الزمنية الأحادية الجدول. ومقارنة تكلفة التخزين لنماذج نظم قواعد البيانات العلائقية الزمنية المتعددة الجداول مع نظم قواعد البيانات العلائقية الزمنية الأحادية الجدول. وكذلك تم تصميم هيكل فهرسة من مزيج عنقودي يلائم الوصول بكفاءة للسجلات المفهرسة بالطابع الزمني. في فصلي النموذج الزمني والفهرسة الزمنية، استعرضنا بوضوح أن حلولنا عملية حيث قمنا برسم معالم التنفيذ الفعال. خوارزميات جديدة للربط المتساوي المتقاطع زمنيا تمت كتابتها. هذه الخوارزميات صممت للتعامل مع نوع خاص من الجداول الزمنية، مثل الجداول الزمنية المتواصلة والمعتمدة علي الحدث.