An open source DBMS for handheld devices

M. Tech Dissertation

Submitted in partial fulfillment of the requirements
for the degree of

Master of Technology

by

Rajkumar Sen
Roll No: 02305035

under the guidance of

Prof. Krithi Ramamritham

Department of Computer Science and Engineering
Indian Institute of Technology, Bombay
Mumbai

Acknowledgments

I am extremely thankful to my guide **Prof. Krithi Ramamritham** for his constant encouragement and invaluable guidance.

Rajkumar Sen
IIT Bombay

Abstract

Lightweight computing devices are becoming ubiquitous and increasing number of applications are being developed for these devices. Many applications e.g E-governance, healthcare etc. need to deal with a significant amount of data. The queries are quite complicated involving selctions, joins and aggregates. Data processing on a handheld device involves many complex issues. Applications like Microbanking involve transactional details. Hence, atomicity and durablity properties need to be satisfied. All such issues necessitate the presence of a local database management system on the device. Also, a DBMS will make application code development easy and fast. However, scaling down the database has not been easy as these devices are constrained by limited stable storage and main memory. Resources in handhelds are increasing with every new device; optimum utilization of these limited resources is a must for such a database system. Evaluation of complex joins and aggregates requires efficient usage of the limited main memory. Memory should be optimally allocated among the database operators. We propose a new data storage model, ID Based Storage which reduce storage cost considerably. We present an exact and a heuristic method for optimally allocating memory among the database operators and compare both the methods. Our query processing strategy guarantees the best query execution plan for any handheld device depending on the available resources. Our lightweight DBMS for handheld devices has been tested on the Simputer.

Contents

Chapter 1

Introduction

Lightweight electronic devices have become ubiquitous and are being equipped with an increasing amount of computing resources. In this era of pervasive computing, computation platforms have extended to small intelligent devices like cellphones, sensors, Smartcards, PDAs etc. As new functionalities and features are being added to these devices, increasing number of applications are being developed. Many such applications need to deal with a significant amount of data leading to the need of embedded database support on these devices [Sma]. Smartcards are being increasingly used in many personal applications like healthcare and address books which need efficient data management. The amount of data is considerable and the queries over these data are more than SPJ queries. Most importantly, the queries have to be locally computed on the device [BBPV00]. Most of the modern day cellphones are being equipped with increasing memory which means more data centric applications are being developed for them. Sensor networks are also coming up rapidly and these collect data from the environment and subject them to various queries. Most of these queries need to be executed on the device itself to reduce communication costs [MFHH02]. Applications for PDAs are dealing with more and more data and regularly execute complicated aggregate queries. Thus, there is an increasing need to facilitate execution of complex queries locally on these lightweight computing devices[KFW+03, Ses99]. Applications like Microbanking and agricultural pricing will involve transactional details. Therefore transaction atomicity and durability are needed to enforce data consistency. In addition. the presence of a DBMS will make application development easy and fast.

However, scaling down the database techniques has not been easy since these lightweight devices come with very limited computing resources. The amount of main memory and stable storage available in such devices is very small. Moreover, resources are not uniform across devices, the computing capability of a cellphone differs from that of a PDA. Optimal utilization of the available resources is mandatory for any system that is developed for such devices.

The limited stable storage accommodates the operating system as well as the database system code, which means very less storage is available to store the data. Storage Models designed for such database systems should reduce storage cost to a minimum to be able to store more data. Limited stable storage precludes the presence of any additional index structures, hence the storage models should try to incorporate some index information in the data model itself. Ideally, index structures which can speed up query processing at no additional storage cost should be maintained.

As far as query execution is concerned, RAM is the most critical resource in these devices. The small amount of memory limits the query processing capabilities in such devices. Existing approaches have used the minimum memory algorithms for every operator which do not give good performance for complex queries involving several joins and aggregates [BPA03]. Query execution time for complex queries can be reduced by using operator algorithms which build in-memory indices and store the aggregate values. The amount of memory available is increasing with every new device. Also, different devices come with different memory sizes. The challenge now is to use the limited memory in the best possible manner. Query execution depends fully on the query execution plan that has been generated by the optimizer. Query plans should be generated in such a way that they make extensive use of the available memory. Optimal memory allocation among the database operators is a must if we need to guarantee the best usage of memory. Memory cognizant query plan generation is a must for efficient execution of a query.

A database management system that can cater to the needs of various forms of handheld devices and use the limited resources efficiently is the need of the hour. The storage schemes of such a system should lead to compactness in the representation of data and indices. Query processing should be able to execute the queries most efficiently using the available resources. Atomicity and consistency also need to be addressed by the system. The database code footprint should be as less as possible. Most of the handheld devices use flash memory for secondary storage. Flash memory reads are equivalent to main memory reads and writes are as costly as disk writes. Hence, the system should minimize writes to secondary storage.

Due to limited storage capability of the handheld devices, it will not be possible to keep all data on the device. Most of the data may be kept in a DBMS that resides on a server. Hence, we need an application that can download the required relations from the server and store them in the DBMS that resides on the device. This means that we also need a synchronizing utility that synchronizes common data between the handheld and the server.

The rest of the report is organized as follows. We discuss the related work in Chapter 2. Chapter 3 covers the storage management issues and introduces ID based Storage, a new storage scheme for small devices. Chapter 4 highlights the issues in query processing and presents the exact memory allocation and our heuristic memory allocation algorithm. In Chapter 5, we discuss the issues that come up during synchronization and chapter 6 deals with transaction management. In Chapter 7, we present our experiments comparing heuristic and exact memory allocation and finally conclude in Chapter 8.

Chapter 2

Related Work

Lightweight versions of some of the popular DBMS like Oracle Lite [OLi], Sybase Adaptive Server Anywhere [Gig01], IBM DB2 EveryPlace [KLLP01] and Microsoft SQL Server for Windows CE [SG00] have been developed for the handheld devices. All these database systems come with reduced code footprint by stripping down database features. However, these systems do not address the issue of utilizing the available memory efficiently. IBM DB2 EveryPlace [KLLP01] has a very simple query optimization model where queries are optimized using predicate information and presence of aggregate operators. Only nested loop algorithm is used to evaluate joins.

Some other related work have taken place in the context of database implementation on smartcards [BBPV00]. It proposes a new storage model called Ring Storage that combines data and index storage in a single structure. Their use of ring indices produces a side effect on query execution since selections become costly. The query processing techniques proposed are quite specific to the smartcard platform where the amount of memory available is extremely small. The algorithms assume the existence of a domain based storage scheme which might not be best storage scheme for all devices. We need to design algorithms that can cater to the needs of various forms of devices.

[BPA03] proposes a query execution model that has a lower bound in terms of RAM usage. The algorithms are mainly nested loop approaches which use minimum memory. To limit the number of iterations, some optimizations called iteration filters are suggested. When more memory is available, the algorithms benefit due to the presence of additional buffers. They take cost of the operators as the number of tuples accessed and distribute the memory on the resulting query plan using a heuristic cost model[1]. They provide guidelines to calibrate the RAM resource of a hardware platform according to the requirements. Their performance study says that their algorithms scale well only for SPJ queries having very few joins. In the presence of several joins and aggregate operators their algorithms do not perform well. For such complex queries, indices are the only alternative to speed up query processing.

[1] The exact allocation algorithm is not mentioned

Chapter 3

Storage Management

Handheld devices are constrained by very limited secondary storage. Hence, the storage model should lead to compactness in the representation of data and indices. In this chapter, we survey the existing storage models, Flat Storage, Domain based Storage [AHK85], and Ring Storage [BBPV00]. We analyse several advantages and disadvantages of each of them. We also propose two new storage models, ID Storage and Bitmap Storage for further compacting data and index representation. We examine and compare all the models based on their storage cost and some properties which impact processing performance.

3.1 Existing Models

3.1.1 Flat Storage

The simplest way to represent relational data is Flat Storage, where the tuples are stored sequentially. The main advantage of this model is access locality. But as discussed earlier, in Flash Memory, sequential access does not perform any better than random access. However, the main disadvantage of this model is that it consumes a lot of space. This is because if any attribute contains duplicate values e.g. Villager.Occupation, then the value Farmer will be repeated for every tuple.

3.1.2 Pointer-based Domain Storage

Our storage model should ensure data compactness, hence duplicate values should be eliminated. If any attribute contains duplicate values, the value should be stored only once. This can be done by partitioning the attribute values into domains which are sets of unique values. This has been suggested in [AHK85][MS83][PVJ90]. This is known as Pointer-based Domain Storage. As shown in Figure 3.1, the tuples reference the values of the attributes by means of pointers. Since one domain can be shared among multiple attributes, this is a very compact structure. However, the complexity increases in the addition, deletion, and updation of tuples as it may lead to insertion and deletion of values in the domain structure.

Another point worth mentioning is that domains can also be used to store variable sized

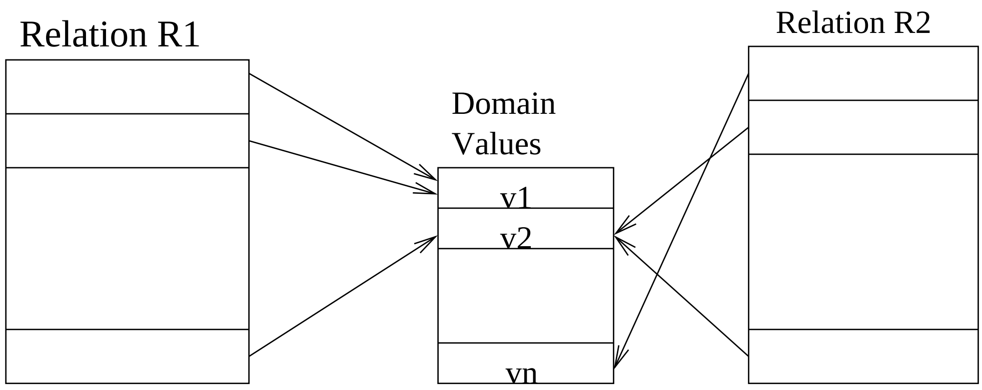

Figure 3.1: Pointer-based Domain Storage

attributes which are generally larger than pointer size. This reduces memory management complexity as there are only fixed size tuples.

3.1.3 Pointer-based Ring Storage

[BBPV00] suggest a modification of Domain Storage that addresses index compactness. Without creating an additional index structure, it uses the domain structure as an index structure. The modification to the simple Domain Storage is that it stores value-to-tuple pointers in the domain structure. It introduces tuple-to-tuple pointers, which connect two tuples that have the same value for an attribute. Thus, an index structure in the form of a ring is created as shown in the Figure 3.2. [BBPV00] also shows how to extend this model to handle join indices.

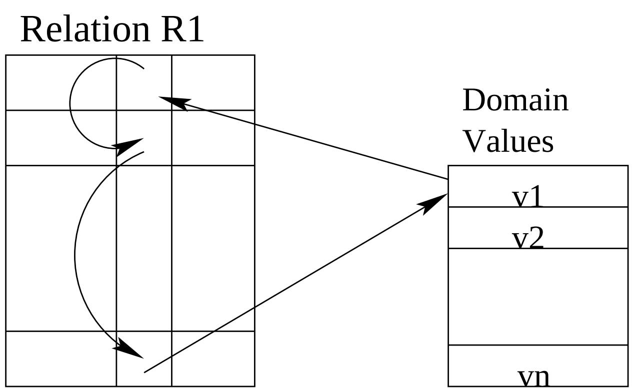

Figure 3.2: Pointer-based Ring Storage

Though the Ring Storage Model addresses data and index compactness, it introduces two new problems

1. For projecting the value of an attribute of a tuple, we need to traverse on an average

half of the ring.

Average size of a ring a is number of tuples of relation R n_r, divided by the number of domain values n_d

$$a = \frac{n_r}{n_d} \tag{3.1}$$

This problem arises because the model tries to integrate the data storage with the index storage. Ideally, we would like to keep the data storage to be unaffected by the index structure.

2. Ring Storage maintains one index per domain. This can be a problem if one domain is shared by multiple attributes (the domain CourseName can be shared by Student.CourseTaken and Professor.CourseOffered). Ideally, we would like to keep an index for attributes that reference the domain.

3.1.4 Some Other Models

Two other models have been suggested by [PVJ90] and [MS83]. Both the models have the same disadvantage as Ring Storage, that of one index structure per domain. The model suggested by [MS83] maintains Domain Trees, hence to answer any simple projection queries, it has to scan all the Domain Trees. The DBGraph model suggested in [PVJ90] maintains tuple-to-value as well as value-to-tuple pointers, hence it can answer simple projection queries easily.

Another storage model proposed by [DVRM99] uses projection index and join index. A projection index contains the copy of a particular column, the column which is being indexed. The columns are removed from the original table and stored separately, with each entry being in the same position as its corresponding base record. This property is called *positional indexing*, and we can access tuples based on their ordinal position. A join index for a foreign key column is a projection index which stores the RowIDs of the matching parent table records instead of the corresponding values. [DVRM99] requires the storage of each foreign key column as join index and the rest of the columns as projection index. It also proposes efficient algorithms for query processing on this model.

3.2 Index Structures

The things that we need to keep in mind while designing our index structure are

- The index structures must occupy as less space as possible.
- The indexing model should be efficient and result in faster querying.

Index structures designed for disk resident data aim to minimize the number of disk accesses and minimize disk space. In most of the cases, data values are repeated in the index structures. Hence, those index structures consume a lot of space. However, when data is present in memory, data need not be stored again in index and only pointers to values can be stored. Memory trees need not have short bushy structure as B-trees since traversing deeper trees is much faster in memory. T-Trees [LC86] have been explicitly designed for memory resident data. [LC86] also evaluates the performance of the index structures. Presently, we do not consider any hash-based or tree-based scheme for our system as they involve storage of additional information thereby consuming a lot of space. However, if the size of the Flash Memory permits storage of an additional index structure, T-Tree can be considered.

3.3 Storage cost evaluation of existing models

We compare the storage cost associated with each of the models. We evaluate the cost for an attribute for which we create a domain. We define the following parameters

N	Cardinality of the Relation
D	Cardinality of the Domain
L	Length of the attribute (expressed in bytes)
p	Size of a pointer (expressed in bytes)
s	Selectivity factor of the domain attribute. $s = \frac{D}{N}$

$$\begin{aligned}
\text{Cost(Flat Storage)} &= N * L \\
\text{Cost(Domain Storage)} &= N * p \quad + \quad D * L \\
\text{Cost(Ring Storage)} &= N * p \quad + \quad D * (L + p)
\end{aligned}$$

The cost equality between Flat Storage and Domain storage becomes $S = \frac{L-p}{L}$. For values of L less than the size of a pointer p, Flat Storage is definitely the winner as it is more compact than Domain Storage. However, for higher values of L, Domain storage becomes more compact. The higher L and lower S, the better is the Domain Storage. Hence, depending on the characteristics of data, the system can decide the storage model for each attribute by doing some estimate of these parameters.

3.4 Extendable ID Based Storage

3.4.1 Basic Strategy

In Domain Storage, we use a pointer of size p, typically 4 bytes, to point to the domain value. This leads to $(p*N)$ bytes being used for the attribute for which we set up a domain structure. Instead of having a pointer to point to the domain value, we can keep a distinct identifier for each of the domain values and store the identifier corresponding to the value in the base relation. The identifier of a domain value can be the ordinal value in the domain table. We can then directly use the identifier value as an offset into the domain structure to retrieve

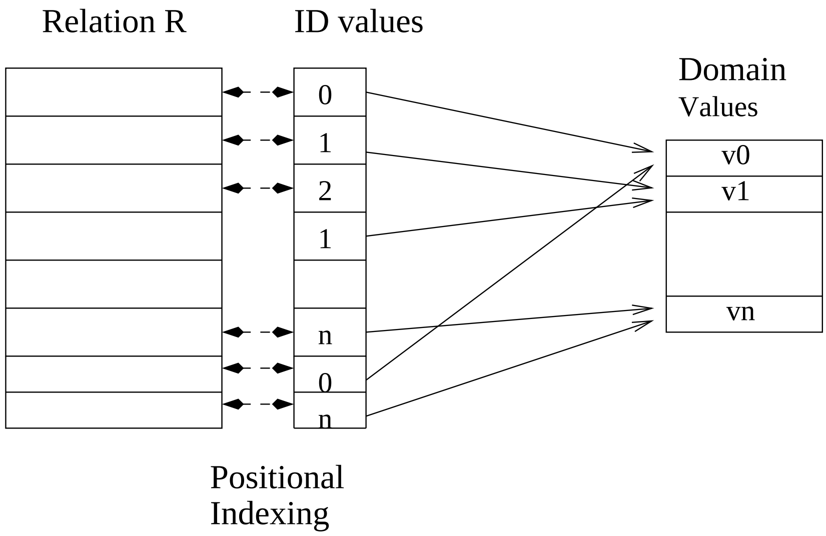

Figure 3.3: ID Based Storage Model

the domain value. Hence, we need not store the ID of the values in the domain structure alongwith the domain values and there is no need for a lookup in the domain structure. A recent work on cache conscious indexing [RR99] uses an integer to represent the IDs. This approach does not lead in saving of space since size of a pointer and a integer for most of the architectures do not differ much, both being typically 4 bytes.

Our storage model (Figure 3.3) uses extendable IDs, where the length of the identifier grows and shrinks depending on the number of domain values. The basic idea behind our model is that D distinct domain values can be distinguished by identifiers of length $\lceil \frac{\log_2 D}{8} \rceil$ bytes. Initially we take 1 byte to represent each identifier. With a single byte we can represent 2^8 domain values. When the number of domain values increases beyond 2^8, one byte will not suffice and we have to increase length of the identifiers by another byte. When the number of domain values exceeds 2^{16}, length of the identifiers is increased to 3. In this way, we increase the length of the identifiers only when it is absolutely required. When the number of domain values exceeds 2^{8l} where l is the length of the identifier, then l needs to be increased. If the deletions result in the number of domain values becoming less than 2^{8l}, then l should be decreased. Thus, the length of the identifier is always $\lceil \frac{\log_2 D}{8} \rceil$ bytes and hence, the data storage cost is reduced by a considerable extent.

The identifiers to the domain values are assigned in increasing order starting from 0 [1] and adding one to the last value every time a new ID has to be assigned. The last identifier value that has been assigned is stored. If the value is one less [2] than 2^{8l}, then the length of

[1]We start from 0 in order to use the ID value as an offset into the domain structure
[2]One less since we start assigning ID values from 0

identifier has to be increased by one byte, otherwise length is not changed. When deletion occurs, we check whether the last ID value assigned was 2^{8l}. If yes, then we decrease the identifier length by one. Tuple creation, insertion and deletion for this model will be slightly more complex than Domain storage. However, this small overhead is more than offset by the space efficiency of the scheme. Tuple insertion always take place at the end of the relation. When a tuple is deleted, we replace it with the last tuple of the relation.

Deletion of Domain values

A problem that exists for domain based models is how to determine whether a domain value has to be deleted. One simple solution is to keep a reference counter alongwith the domain value. But such a strategy defeats the very purpose of saving more space. Instead, we do a periodic check whether all the domain values are referenced by reading the relations. But how often should the check be performed. We keep a single byte count counting the number of deletions done since last time the check was done. Staring with zero, for every deletion, the counter is increased by one. If the value exceeds a threshold $\frac{1}{S}$, S being the selectivity of the domain attribute, it means that there is a chance of a domain value being deleted. The check is done and the value of counter is reset to 0. For small values of $\frac{1}{S}$, we keep the threshold as a small multiple of $\frac{1}{S}$.

Ping Pong Effect

We observe that at the boundaries when the length of the identifiers increases, there is reorganization of the ID values. When the length decreases, again there is reorganization. If there are frequent insertions and deletions at these boundaries, there might be a Ping Pong effect, resulting in a lot of reorganization of the data. Such a situation should be avoided.

One way to avoid frequent reorganization at the boundaries is to stop deletion of tuples at the boundaries. When the length of the IDs increases, we suspend deletion. Instead, we mark the deleted tuples as holes till their number crosses a threshold. All tuple additions then take place in the holes. When the holes are exhausted, we again start inserting at the end.

Projection Index

Since the length of the identifiers grow and shrink dynamically, a lot of reorganization would take place in the relation data. It would make more sense to project out the domain attribute identifier value from the rest of the relation and stored separately, with each identifier being in the same position as its corresponding base relation. This property also called *Positional Indexing* enables us to access tuples based on their ordinal position. Thus, the IDs behave as a *Projection Index* for that domain attribute. Data reorganization due to extendable IDs is avoided on the whole relation data and is restricted to the projected column only whose size is much lesser than base relation data. Note that a similar strategy is used in [DVRM99].

3.4.2 Primary Key-Foreign Key Relationship

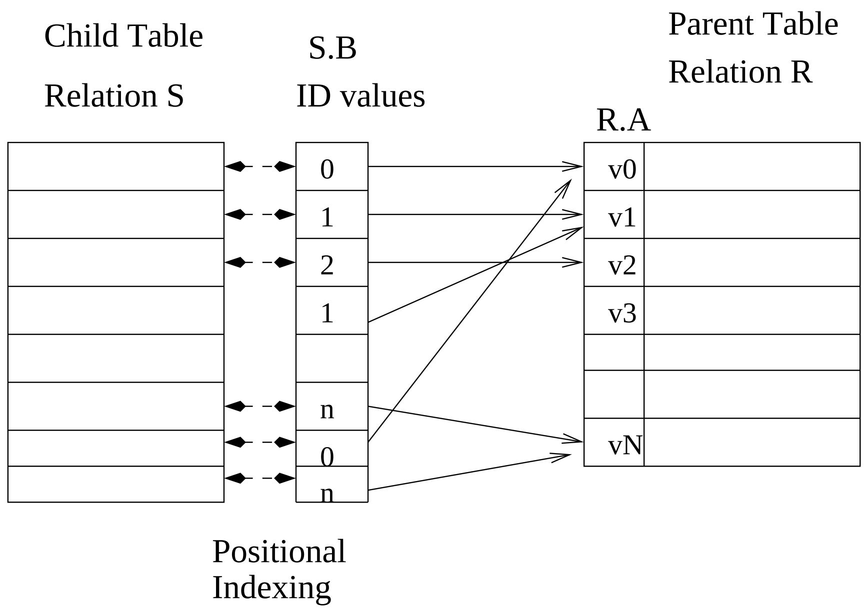

Figure 3.4: Join Index for Primary key-Foreign key Relationship

A primary key is a domain in itself since there are no duplicates. Although a primary key will not be stored in a domain structure and will be stored in Flat Storage, we can treat the primary key as a domain for the foreign key and implement the ID based scheme to represent the foreign key values. The foreign key values now reference the primary key values by their corresponding IDs. The projected foreign key column now becomes a *Join Index* for the child table and the parent table since for every foreign key value we have the information of the matching parent table tuple. This leads to efficient join of the parent and the child table. Figure 3.4 illustrates the join index for relations R and S where R.A is the primary key and S.B is the foreign key. Note that the situation resembles *Join Index* in [DVRM99].

3.4.3 Storage Cost

We compare ID Storage model with Domain Storage and examine the conditions under which ID Storage saves more space than Domain Storage.

$$\text{Cost(Domain Storage)} = N * p + D * L$$
$$\text{Cost(Id Storage)} = N * \lceil \frac{\log_2 D}{8} \rceil + D * L$$

The cost equality between the two models become $p = \lceil \frac{\log_2 D}{8} \rceil$. The size of the pointer p is typically 4 bytes. We observe that when the number of domain values is not greater than $2^{8(p-1)}$ (2^{24} for $p = 4$), ID Storage is always a winner over Domain Storage. For a handheld device which deals with very less amount of data, the cardinality of the relations will not be tremendously high . Hence, we can easily expect the condition to be true for majority of the relations. We have already seen the conditions in which Domain Storage wins over Flat Storage. When ID Storage outperforms Domain Storage, it can be easily seen that it outscores Flat Storage as well.

3.4.4 Insertion, Updation, and Deletion

In section 3.4.1 we have already mentioned the basic scheme for inserting and deleting values. In this section, we present a few more details.

- Insertion

  When a tuple is inserted, we always append the tuple at the end. For the attribute that contains a domain structure, the ID of the value also has to be appended at the end of the ID value structure in order to maintain *Positional Indexing*. We first lookup the value in the domain structure and if present, store the corresponding ID value. If the domain value is not present, we append the domain value at the end in the domain structure and get a new ID for it. The conditions under which length of identifier increases that were mentioned in 3.4.1 are checked.

- Deletion

  When a tuple is deleted, we replace it with the last tuple of the relation and truncate the relation. The ID value structure is modified in the same manner, i.e, last tuple's ID value is shifted to the hole created. This strategy keeps the deletion code simple. Another way of performing deletion could be to keep track of the holes and while inserting a tuple, insert it in any available hole. If no hole is available, append at the end.

  Another problem that occurs for domain based models is how to determine whether a domain value has to be deleted. One simple solution is to keep a reference counter alongwith the domain value. But such a strategy defeats the very purpose of saving more space. Instead, we do a periodic check whether all the domain values are referenced or not by reading the relations. But how often should we perform the check. We keep a single byte count regarding the number of deletions done since last time the check was done. Staring with zero, for every deletion, the counter is increased by one. If the value exceeds a threshold $\frac{1}{S}$, S being the selectivity of the domain attribute, it means that there is a chance of a domain value being deleted. We have to do the check and reset the value of counter to 0. For small values of $\frac{1}{S}$, we keep the threshold as a small multiple of $\frac{1}{S}$.

- Updation

  When a tuple is updated, we need to check if the domain value also changes. If yes, then we have to repeat the procedure in insertion and deletion for domains. If the domain value is new, we need to add it to the domain structure and get a new ID for it. We also need to increment the global tuple deletion counter since updation means one reference to the replaced domain value has been deleted.

3.4.5 Advantages

ID based Storage has the following advantages

- Instead of storing a pointer (of size 4 bytes), the model uses extendable IDs. A lot of space is saved especially when the number of domain values is less. For most of the relations that would reside in a handheld, the cardinalities will be such that the number of domain values lie between 2^8 and 2^{16}.
- Using the same scheme, we can represent primary key-foreign key relationship and get a join index almost for free. This leads to efficient join of the parent and the child table.
- In this scheme, ID values are separately stored than the base relation data. When we read the relation or when we need to update, increase/decrease the length of the ID values, the projection index of ID values have to be read into memory and then updated. This may seem to be a potential overhead. However, we see that due to very small size of the projection index, there will be very less cache misses. The number of IDs that will fit in one cache line is $\frac{C}{l}$ where C is the cache line and l the length of the IDs. Considering $C = 32$ and $l = 2$, 16 IDs fit into one cache line.

3.5 Bitmap Based Storage

3.5.1 Basic Strategy

We had previously surveyed the Ring Storage model proposed in [BBPV00]. The Ring Storage model combines data storage as well as index storage into a single structure at a slight additional cost. The index associated with Ring Storage is a selection index. Ring Storage also defines a join index for primary key-foreign key pair.

In disk-based systems, bitmap indices are used for efficient querying on multiple keys. A bitmap index is built on a single key. Bitmap indices consume less space when the number of distinct values for which the bitmaps are created is small. We observe that from a bitmap, it is possible to retrieve the tuples of the relation. This forms the basic idea behind Bitmap Based Storage.

In this storage model, a bitmap index is created on the domain attribute. A bitmap is created for each value that the domain attribute can take. Each bitmap has as many bits equal to the number of tuples of the relation. Hence, it serves as an index for the domain attribute. The main difference between the disk-based bitmap indexes and our proposed model is that the values of the domain attribute are not stored in the base relation. The indices can be used to retrieve the domain values associated with each tuple. Suppose we need the domain attribute value for tuple i. We search examining the bitmaps of each domain value and stop when the ith bit is set. Hence, this scheme also serves as a data storage model. Figure 3.5 shows the bitmap indices for the domain values[3] of Relation Medical Visit Info.

[3]Note that letters A,B,C have been used to name the doctors. In general, these names will be of greater length and hence candidate for domain

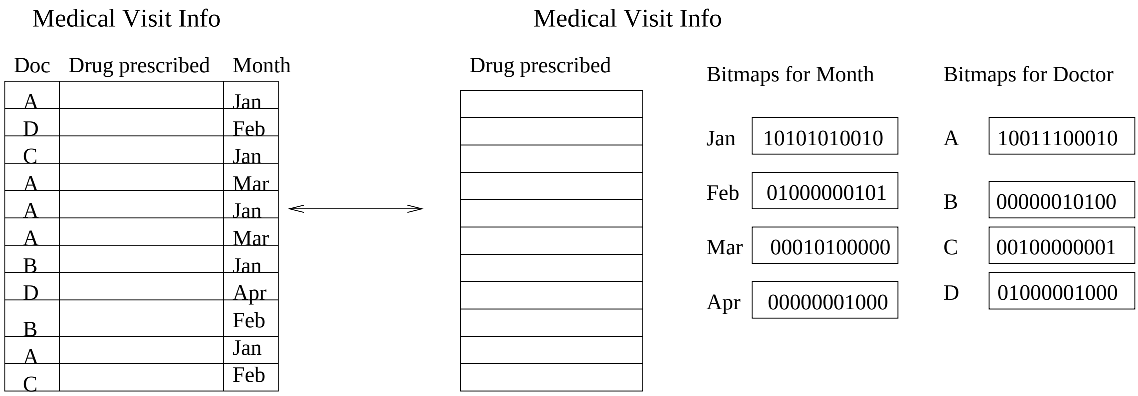

Figure 3.5: Bitmap Based Storage

Bitmaps act as a selection index for the domain attributes, and is the most beneficial when the query is a selection on multiple keys. However, like Ring Storage, an additional cost has to be borne for projecting out a tuple to its corresponding domain value since we need to traverse on an average half of the bitmaps associated with the domain values. All the problems associated with bitmaps appear here also thus making code for modification to these bitmaps complex. The conditions that need to hold in order for bitmaps to be efficient is that the number of domain values is very less compared to the cardinality of the relation and there is a need of efficient querying on multiple attributes e.g. if we examine medical database of a person, number of doctors visited would be very less compared to the total number of visits and he might need answer of a query like "How many times in Jan he visited Dr. A?"

3.5.2 Primary Key Foreign Key Relationship

In Ring Storage, alongwith the values in parent table a pointer to one of the values in child table is maintained. A similar approach can be applied in case of Bitmap Storage also. Alongwith each parent table we store a bitmap. These bitmaps form a join index since for every parent table value, we have the information of the matching tuples of the child table. Since the size of the bitmaps grow and shrink, it may make more sense to project out these bitmaps and store separately from the parent relation, maintaining *Positional Indexing*.

3.5.3 Comparison with Ring Storage

We compare Bitmap Based Storage and Ring Storage and examine the conditions in which Bitmap wins over Ring.

$$\text{Cost(Ring Storage)} = N * p + D * (L + p)$$
$$\text{Cost(Bitmap Storage)} = D * (L + \lceil \tfrac{N}{8} \rceil)$$

The cost equality condition becomes $p = \frac{\lceil \frac{N}{8} \rceil}{1+\frac{1}{S}}$. Bitmap based Storage wins over Ring for those relations where the number of domain values is very less compared to the number of tuples, i.e, selectivity S of the attribute is low. For relations which do not satisfy the bitmap constraints, Ring Storage is definitely the winner. However, indexing capabilities of both the models are different.

Another factor that has to be considered in this comparison is the size of the intermediate structures while executing a select query. In case of a select query on multiple attributes, the number of bit vectors needed by bitmap based storage is equal to the number of attributes that form part of the selection. The bit vectors will be stored in memory which is a limited resource and too many bit vectors might nullify the advantage of using the scheme.

We earlier proved that ID Storage performed better than Domain Storage for majority of the cases. But we cannot claim that Bitmap Storage is better or worse than Ring Storage. Both have their own utility and the final decision about the storage model to use has to taken by the system depending on the relation characteristics and expected type of queries.

3.6 Performance Properties

In addition to the storage cost, we also need to look at the properties[GH90] that affect the performance of the storage manager. These properties are listed below

1. Cost to insert a tuple

   Insertion in a relation always takes place at the end of the relation. Flat Storage has the most simple and efficient code since only one relation file has to be manipulated. Insertion in Domain and ID Storage is more complex than Flat Storage. Insertion in Ring Storage and Bitmap Storage are much more complicated since they involve manipulation of ring pointers and bitmaps respectively.

2. Cost to delete a tuple

   Deletion is the easiest in Flat Storage, a bit complex in Domain and ID Storage, and most complex in case of Ring Storage and Bitmap Storage. A lot of ring pointer manipulations need to be done in Ring Storage. In Bitmap Storage, the length of the bitmaps change.

3. Cost to update a tuple

   Tuple updation is again cheapest in Flat Storage. For Domain and ID Storage, it gets complicated since new values could be added to domain and old values deleted. Ring pointers and bitmap changes are the complexities involved in Ring and Bitmap storage.

4. Cost to project an attribute

   Again, Flat Storage is the winner. Projection in Domain and ID Storage is easy since we have directly the information about the domain value. Projection is most difficult in Ring and Bitmap Storage.

5. Cost to select a tuple

   Ring Storage turns out to be the best for single attribute selection. When queries are selections on multiple attributes, Bitmap Storage is the most efficient. Selection of tuples is inefficient in Flat, ID, and Domain storage models since they do not involve any index structure.

3.7 How to choose the best storage model

We present a summary of the conditions in which a storage model would perform poor, good and best. These would serve as a guideline for the storage manager to decide which storage model to be used.

Storage Model	Good	Poor	Best
Flat Storage	$S > \frac{L-p}{L}$	$S < \frac{L-p}{L}$	Updates are very frequent
Domain Storage	$S < \frac{L-p}{L}$	$S > \frac{L-p}{L}$	$p < \lceil \frac{\log_2 D}{8} \rceil$,updates are few
ID Storage	$S < \frac{L-p}{L}$	$S > \frac{L-p}{L}$	$p > \lceil \frac{\log_2 D}{8} \rceil$,updates are few
Ring Storage	$p < \frac{\lceil \frac{N}{8} \rceil}{1+\frac{1}{S}}$	$p > \frac{\lceil \frac{N}{8} \rceil}{1+\frac{1}{S}}$	Frequent selection on single attribute
Bitmap Storage	$p > \frac{\lceil \frac{N}{8} \rceil}{1+\frac{1}{S}}$	$p < \frac{\lceil \frac{N}{8} \rceil}{1+\frac{1}{S}}$	Frequent selection on multiple attributes

Chapter 4

Query Processing

Lightweight computing devices are characterized by a small amount of main memory and most of them use flash memory as secondary storage. Writes to flash memory are very costly. Hence, the query processing schemes should minimize materialization in secondary storage. Flash memory reads are equivalent to main memory reads, so there is no need of read buffer. If read/write ratio is very high we can use flash memory as write buffer, otherwise main memory should be used as write buffer. The main memory limits the query execution capabilities in a small device, hence the techniques should make optimum usage of the memory.

4.1 Query Plan

The query processing unit should minimize writes to secondary storage. Hence, materialization of intermediate results should be minimized. Materialization if absolutely necessary, should be done in main memory. All these constraints favour the use of left-deep tree as the query plan tree since all other query plans resort to some materialization. In left-deep tree, the right operand of every operator is one of the initial relations appearing in the query. Left-deep tree is most suited for pipelined evaluation, since the right operand is always a stored relation, and thus only one input to each operator is pipelined.

4.1.1 Need for optimal memory allocation

When we have a fully pipelinable schedule, we cannot assume that the entire memory is available for all the operators[HSS00]. This problem is crucial since handhelds are constrained by very less memory. Memory has to be shared optimally among all the operators. Of course, it all depends on how each database operator is evaluated, how much memory each requires. If the nested loop approach is taken, we can guarantee a fully pipelinable schedule with minimum memory usage. Nested loop joins involves as many scans over the inner relation as there are tuples in the outer relation, aggregation involves as many scans over the input as there are distinct aggregate values. Though nested loop approach guarantees the minimum usage of memory, it does not give the best query execution plan. Query plans depend on the amount of main memory available for storing indices and aggregate values. Memory usage need not be restricted to a bare minimum at the cost of performance. The amount of memory available in handhelds is increasing with every new device, more important is the fact that different handhelds come with different memory sizes. A database system for handheld devices should be able to cater to the needs of various forms of handheld devices and exploit their resources as efficiently as possible, therefore we cannot restrict our strategies to a static set. Nested loop join need not be the only join technique used, memory might be available to perform a hash join which gives better performance. We need to have a strategy that guarantees the best query execution plan for every device depending on available main memory.

4.2 Query optimization for complex queries

Existing approaches to query processing in small devices[BBPV00, BPA03, KLLP01, SG00] use minimum memory nested loop algorithms for every operator thus minimizing the usage of memory. Such an approach gives very poor performance for queries involving complex joins over several relations and aggregations. The cost of a query execution plan depends on the operator algorithms used and ordering of the operators. Execution time for complex queries can be reduced by using non-nested loop operator algorithms that gives better performance. The join order also plays a crucial role since it determines the size of the subtree over which the iterators reiterate.

Consider the healthcare database schema [BBPV00] and some sample queries in Figure 4.1.

Healthcare Database Schema	Doctor *(DocId,name,speciality,...)* Prescription *(VisitId,DrugId,qty...)* Visit *(VisitId,DocId,date,diagnostc,..)* Drug *(DrugId,name,type,...)*
Query Q1.	*Who prescribed Antibiotics in 2003?*
Query Q2.	*Number of Antibiotics prescribed per doctor in 2003.*
Query Q3.	*Number of prescriptions per type of drug*
Query Q4.	*Number of prescriptions per doctor and per type of drug*

Figure 4.1: Sample schema and queries

Query Q1 is a complex join over several relations while Q2 to Q4 are aggregate queries. Query Q4 is the most complex since the result should be grouped on two attributes. Consider the query plans for Q1 and Q4 in Figure 4.2 and 4.3. Alongwith every operator is given the size of its result. We observe that QEP 3 contains a cartesian product and is the worst plan. QEP 1 and QEP 2 do not have a cartesian product but the cost differs due to different join order. QEP 2 is the best plan for query Q1. If we consider the aggregate query Q4, the cost of the QEP 1 and QEP2 vary a lot depending on the GroupBy operator at the root. If there is not enough memory to store the aggregated values and GroupBy needs to be evaluated in a nested loop manner, then the cost of the subtree below GroupBy should be minimized to get the optimal join order. Memory should be optimally allocated among the GroupBy and Join operators to get the most efficient QEP. Query optimization is a must for such complicated queries.

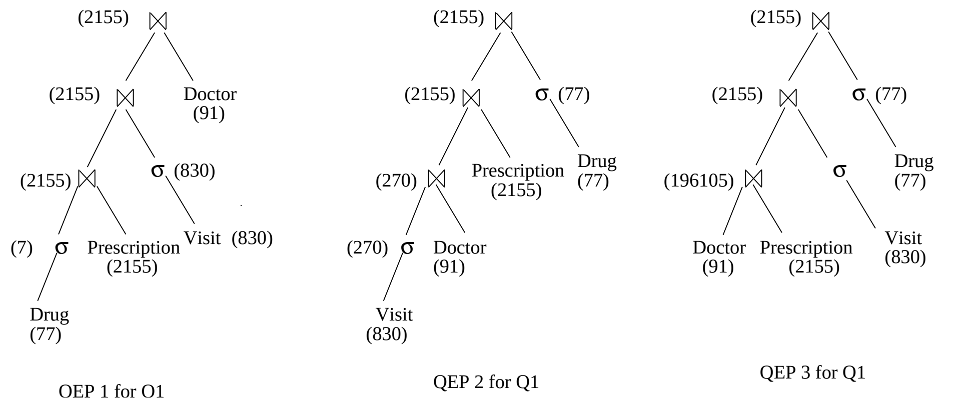

Figure 4.2: Query execution plans for Q1

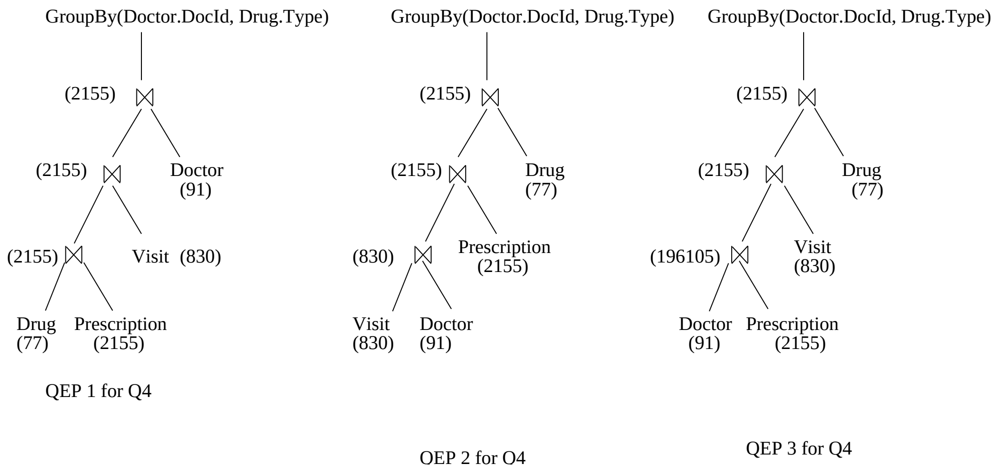

Figure 4.3: Query execution plans for Q4

4.3 Cost based query optimization

Our query optimizer is cost-based since only cost-based approach can ensure the best execution plan. The query execution plan takes the shape of a left-deep tree since it is most suited for pipelining. The optimal join order is evaluated using the System-R dynamic programming approach [SACL79]. Estimation of the size of the result of an operator is done using the rules given in [SSK02].

We also consider the heuristics employed in traditional optimizers. One such heuristics is pushing selections down. When we have left-deep tree and a fully pipelinable schedule, then pushing down selections become tricky. Selections which occur at the leaf nodes of the left-deep tree reduce the number of tuples going up the tree. Other selections which occur in the right child of a join are evaluated as many times as the join attribute itself. This is because unlike disk based systems we do not memory to materialize the selected tuples. Non-leaf selections need to be evaluated in the join algorithms together with the join filters.

4.4 Operator Evaluation Schemes

In this section, we present the schemes for evaluating the database operators. All the schemes for an operator use different amount of memory and have different cost. All the schemes take care of the fact that left-deep tree is used as query plan. The memory usage of the schemes excludes the input buffers and the output buffer. Cost of a scheme is the expected time of computation. Evaluating computation time involves considering the number of tuples read and index, aggregate list creation, and lookup[DKO$^+$84]. Relations R and S are the inputs to the operators with m and n tuples respectively, R being the input to the unary operators. We also define the following parameters

p: size of a pointer
L: average length of join attribute of R
$comp$: time to compare the join keys.
b: number of hash buckets. determined by the hash function
f: hash table fudge factor
$hash$: time to hash a key.
$write$: time to write in memory.
$read$: time to read from memory [1]

4.4.1 Selection and Projection

The cost of selection varies depending on the type of storage model and index available.

No index

Selection and Projection do not need any additional memory apart from the current tuple buffer and require one pass over the relation.

Scheme	Cost	Reason
SelectionNoIndex	$m * (read + comp)$	Read and Compare every record
ProjectionNoIndex	$m * read$	Read every record

Scheme	Memory	Reason
SelectionNoIndex	0	No extra memory
ProjectionNoIndex	0	No extra memory

Ring index and Bitmap index

Selection and Projection with a bitmap index and ring index have different costs. Memory usage for bitmap model is the size of the bit vectors. Suppose k number of attributes are part of the selection/projection. Following the notations used in the previous chapter, D_i is the number of domain values for attribute i. Please note that we have assumed number of tuples to be N to maintain the notations used in the discussion on storage models, hence $m = N$.

[1]Note that read from flash and main memory takes almost the same time

Scheme	Cost	Reason
SelectionRing	$\sum_{i=1}^{k} read * (\frac{D_i}{2})$ $+ \sum_{i=1}^{k} read * (\frac{N}{D_i})$	For every attribute, read ring values,(no order in ring, on an avg. read half of the values) Traverse the ring
SelectionBitmap	$\sum_{i=1}^{k} read * (\frac{D_i}{2})$ $+ \sum_{i=1}^{k} read * (\frac{N}{D_i})$	For every attribute, read the bitmaps ,(no order assumed, on an avg. read half of the bitmaps) Read the selected tuples
ProjectionRing	$N * \sum_{i=1}^{k} read * (\frac{\frac{N}{D_i}}{2})$	For every tuple, traverse half a ring
ProjectionBitmap	$N * \sum_{i=1}^{k} read * (\frac{\lceil \frac{N}{8} \rceil}{2})$	On an avg. read half of the bitmaps

Scheme	Memory	Reason
SelectionRing	0	No extra memory
SelectionBitmap	$k * (\lceil \frac{N}{8} \rceil)$	Size of the bit vectors
ProjectionRing	0	No extra memory
ProjectionBitmap	0	No extra memory

4.4.2 Join

In a left-deep tree join strategy, the right input R is a always a base relation and the left input S is pipelined. Hence, we consider only nested loop join and hash join. The following are the join schemes and their costs[2] and memory usage

Nested Loop Join

Nested loop join does not require any temporary memory structures except for the two tuple buffers.

Scheme	Cost	Reason
NLJ	$(m * n) * (read + comp)$	Read and compare in nested loop manner

Scheme	Memory	Reason
NLJ	0	No extra memory needed

[2]We do not consider the time to read the left relation that is being pipelined since that cost is common to all schemes

Indexed Nested Loop Join

In this scheme, we create an in-memory index on the right input relation R and probe into it when we have a tuple from S. We store the pointers to the tuples in the relation as well as the key values in the index. We have three choices as an index (i) array index (ii) linked list (iii) binary search tree. We do not consider complex index structures like B-tree and T-tree.

Array indexes are the most efficient but the main drawback is the shifting of values that take place when the index is created.

Scheme	Cost	Reason
ARRINLJ	$m * (read + write)$ $+\sum_{i=1}^{m}(comp * \log i)$ $+\sum_{i=1}^{m}(read + write) * i)$ $+n * (comp * \log m + read)$	reading the tuple and writing the index Lookup during index creation(worst case) Shifting during index creation(worst case) Lookup into the index

Scheme	Memory	Reason
ARRINLJ	$(p + L) * m$	The value and the pointer to the tuple

We observe that the cost is $O(m^2)$.

When we use linked list, the lookup during index creation as well as during probing later becomes costly since we need to traverse on an average half of the index. Complexity remains $O(m^2)$ and size of index also increases. Hence, we reject linked lists.

A binary search tree requires two extra pointers, but it provides better cost.

Scheme	Cost	Reason
BSTINLJ	$m * (read + write)$ $+\sum_{i=1}^{m}(comp * \log i)$ $+n * (comp * \log m + read)$	reading the tuple and writing the index Lookup during index creation Lookup into the index

Scheme	Memory	Reason
BSTINLJ	$(3p + L) * m$	Two extra pointers for left and right child

We observe that the term for lookup during creation $\sum_{i=1}^{m}(comp * \log i)$ can be approximated to $comp * m \log m$. We select binary search tree as the indexing technique since cost of ARRINLJ is $O(m^2)$ and that of BSTINLJ is $O(m \log m)$.

Hash Join

In this scheme, we create a hash table for the right input relation R. The hash table only contains pointers to the tuples and not the key values[3]. When a tuple from S arrives, we hash it and use the hash value to join with all candidate tuples of R.

[3]Since we are not storing key values in hash table, the number of lookup values to check will be $\frac{m}{b}$

Scheme	Cost	Reason
HJ	$m * (read + hash + write)$ $+n * hash$ $+n * (\frac{m}{b}) * (comp + 2 * read)$	reading the tuple and writing the index hashing into the table traversing the hash bucket

Scheme	Memory	Reason
HJ	$(2p * m) * f$	Pointers to the tuple and to the next bucket element

The values of $read, write, comp$ will be almost same since they all deal with data in memory. Considering $read = write = comp = K$, the cost expressions of the schemes simplify to

Scheme	Cost
NLJ	$(m * n) * (2K)$
INLJ	$m * (2K + K * \log m) + n * (K * \log m + K)$
HJ	$m * (hash + 2K) + n * (hash + (\frac{m}{b}) * (3K))$

4.4.3 Aggregation

In this section, we present two schemes for aggregation, nested loop scheme and buffered scheme. For the following schemes, we assume that the input is R. We introduce some more parameters

d: number of distinct groupby values
$aggr$: time to do the aggregation for one tuple
val: size of the aggregation value
sub: cost of the subtree below the aggregate operator in the query plan

Nested Loop Aggregation

In this scheme, we perform as many iterations over the input as there are distinct groupby values, and one initial iteration to get the value to start with. At each iteration, the aggregation for the current value is done and the next value to be aggregated is determined. We need to store in memory only the current value to be aggregated and the next value, hence memory requirement is nil. When the aggregate operator is sort, then at each iteration instead of aggregating the tuples, we need to output them.

Scheme	Cost	Reason
NLASum	$(1 + d) * m * (read + aggr)$ $+ (d * sub)$	Reading the input in nested loop manner

Scheme	Memory	Reason
NLASum	0	No additional memory

Note that cost and memory for NLAAvg and NLASort are the same as NLASum[4].

Buffered Aggregation

In this scheme, we iterate over the input only once and allocate memory for the results. We construct the aggregates on the fly.

Scheme	Cost	Reason
BASum	$m * (read + aggr)$	Reading the input only once

Scheme	Memory	Reason
BASum	$(d * val)$	Memory to store all aggregate results

The cost and memory for BAAvg is the same as BASum. Note that there is no Buffered Aggregation scheme for Sort operator since the size of the results is equal to the size of the relation.

The database operators have been implemented using the *Iterator Model* [Gra93]. In this model, each operator is an iterator that supports three procedure calls: *open* to prepare an operator for producing an item, *next* to produce an item, and *close* to perform final clean-up. A QEP is activated starting at the root of the operator tree and progressing towards the leaves. The dataflow in the model is demand-driven: a child operator passes a tuple to its parent node in response to a next call from the parent.

4.4.4 Benefit/Size of a scheme and operator cost function

In the previous sections, we defined several schemes for implementing the database operators and their associated cost and memory. Every scheme is characterized by a benefit/size ratio, which represents the benefit of the scheme per unit memory allocation. In this section, we describe how to compute this ratio for every scheme.

Suppose there are n schemes $s_1, s_2......s_n$ to implement an operator o. All these schemes have a cost and memory associated with them. The cost function of an operator is the collection of (memory,cost) points of its schemes Figure 4.4b. It is a piecewise linear function. We define the function *min*, which for every operator returns the scheme that has the maximum cost and minimum memory.

$$\begin{aligned} &min(o) = s_{min} \\ &\forall i, 1 \le i \le n : Cost(s_i) \le Cost(s_{min}), \\ &\qquad Memory(s_i) \ge Memory(s_{min}) \end{aligned}$$

Suppose s_{min} is the scheme returned by min for operator o. Then the benefit and size of every scheme s_i is defined by

$$\begin{aligned} &Benefit(s_i) = Cost(s_{min}) - Cost(s_i) \\ &Size(s_i) = Memory(s_i) - Memory(s_{min}) \end{aligned}$$

[4]For average, we need to maintain a count

An operator is defined by the benefit and size of each of its schemes. Every operator is a collection of (benefit,size) points, n points for n schemes[5]. Figure 4.4a shows the benefit and size points for an operator. In general, the benefit of a scheme increases with the amount

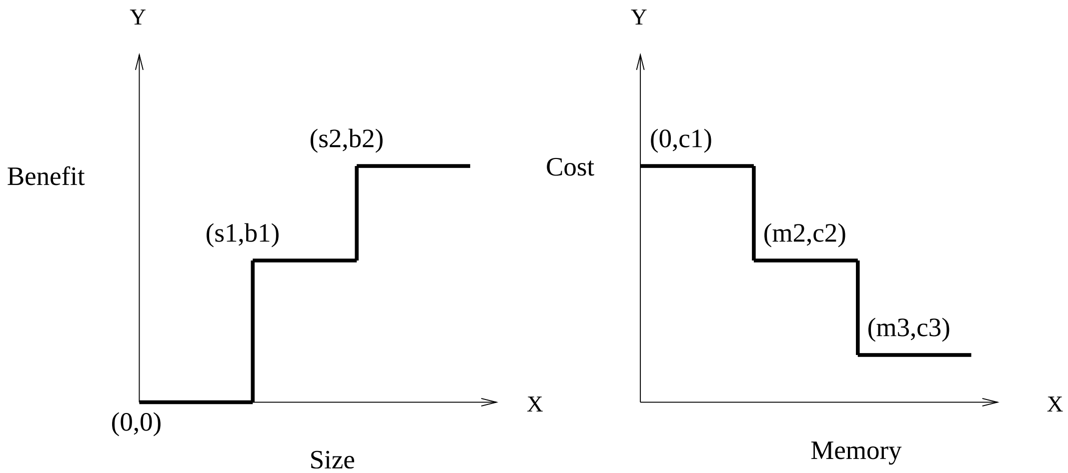

Figure 4.4: (a) Benefit,Size points for an operator (b) Operator cost function

of memory provided. If $Memory(s_i) < Memory(s_j)$, we expect that $Cost(s_i) > Cost(s_j)$ although it may not be the case always.

1-Phase and 2-Phase Memory Allocation

In the 1-phase approach, the query optimizer is made to be memory cognizant. The optimizer takes into account division of memory amongst operators while choosing between equivalent plans and . In the 2-phase approach, the query is first optimized and an optimal query plan is found. Then, the division of memory amongst the operators is done. The 2-phase approach is able to optimally divide the available memory amongst the operators of a given plan, the plan being considered may itself be suboptimal for the available memory. Hence, ideally the 1-phase approach should be taken. However, 1-phase approach means the optimizer becomes highly complex.

In the traditional 2-phase approach, maximum memory is allocated to all the operators while determining the optimal plan in phase 1. For every operator in the plan, the scheme is determined in phase 1 e.g. nested loop for join operator. Memory allocation in phase 2 is based on the cost functions of these schemes. Such a strategy assumes that memory is available for all the operator schemes which may not be true for a resource constrained device limited by main memory. Some of the schemes may not be *possible*[6]. For a limited memory device, depending on the amount of available memory, we need to determine the best scheme for every operator out of all *possible* ones. Schemes chosen in phase 1 and schemes after phase 2 need not be the same. Optimal division of memory to the operators will involve the decision of selecting the best *possible* scheme for every operator given the available memory.

Our query optimizer is a 2-phase optimizer with a modification. In phase 1, it determines the join order by choosing the least cost scheme for every operator. However, in phase 2, it

[5]Point for s_{min} is (0,0)

[6]A scheme is *possible* if its memory requirements can be met with the available memory

chooses the best *possible* scheme for every operator depending on the memory allocated to it.

4.4.5 Exact Memory Allocation

[HSS00] propose a 2-phase exact solution to the memory allocation problem. In phase 1, the scheme for every operator is determined. In phase 2, the cost functions of all the operator schemes are used to divide the memory among all operators. For optimal allocation of memory among n operators, it merges the cost functions of two operators and forms a superoperator cost function. The superoperator cost function stores the optimal division of memory along with the cost value for each memory division point. Then this combined cost function is merged with the cost function of the third operator to get combined cost function of the three operators. In this way, all operator cost functions are merged into a single cost function and then memory division is done by tracing back the steps for each intermediate plan.

The exact solution can be used in our context with some modifications. The schemes for every operator should be determined in phase 2, hence memory need to be divided among the operator cost functions rather than the cost function of an operator scheme. The operator cost functions are piecewise linear functions and hence we can use the exact memory allocation algorithm by replacing the scheme cost functions with the operator cost functions in the algorithm 4.2 of [HSS00]. The number of straight line segments of an operator cost function is equal to the number of schemes for that operator. The scheme for an operator is decided based on the memory allocated to it. The amount of memory allocated to an operator will exactly match with one of its schemes. This is guaranteed by Theorem 4.2 in [HSS00] which states that optimal division of memory for piecewise linear functions takes place only at change-over points.

Time and Space Complexity

Suppose there are n_i schemes for an operator i. The time complexity of merging two operators i and j is $O(z^2 \log z)$ where $z = max(n_i, n_j)$. The number of line segments in the combined cost function is $O(z^2)$ in the worse case.

Time complexity $=\sum_{i=1}^{m}((z)^2)^i \log((z)^2)^{i-1}$ where m is the number of operators.

A superoperator point consists of four values cost, memory and the division of memory at that point. Assuming an integer to store them and I as integer size,

Space complexity $=(4 * I) * \sum_{i=1}^{m-1}((z)^2)^i$.

4.4.6 Heuristic Memory Allocation

The time and space complexity of the exact algorithm is very high. For values of $z = 3$ and $m = 4$, the algorithm requires space of around 96k [7]. Some of the handheld devices might not be able to allocate so much memory. We need a heuristic solution to the memory allocation with a reduced time and space complexity.

The heuristic solution works as follows. Given the available memory, we use a heuristic to determine which operator gains the most per unit memory allocation and allocate memory to that operator. The gain of every operator is determined by its best possible scheme. We repeat this process until all the operators have been allocated memory. Our heuristic is as follows:

[7] Considering 4 bytes for integer

$MemAllocate(M_{total})$
{Evaluate the minimum memory required to execute the query plan
1. $M_{min} = \sum_{i=1}^{m} Memory(min(i))$
{Allocate initially the minimum scheme
to all the operators}
2. for $i = 1$ to m do
3. $Scheme(i) = min(i)$
{Evaluate the available memory for allocation}
4. $M_{avail} = M_{total} - M_{min}$
5. $RemoveSchemes(M_{avail})$
6. $s_{best}, o_{best} = GetBestScheme(M_{avail})$
{s_{best} is the best scheme and o_{best} is the corresponding operator}
7. if no best scheme $return$
8. else
{Allocate $Memory(s_{best})$ to the operator o_{best} }
9. $M_{avail} = M_{avail} - Memory(s_{best}) + Memory(Scheme(o_{best}))$
10. $Scheme(o_{best}) = s_{best}$
11. $RemoveSchemes(o_{best}, s_{best}, M_{avail})$
12. $RecomputeBenefits(o_{best}, s_{best})$
13. end if
14. goto step 6

Figure 4.5: Pseudo Code for $MemAllocate$

Select the scheme that has the maximum benefit/size ratio and allocate its memory.

The procedure $MemAllocate$ that allocates the memory is shown in Figure 4.5. It takes as input the memory available to evaluate the query M_{total} and allocates memory to all the operators. It makes use of two procedures $GetBestScheme$ (Figure 4.6) and $RecomputeBenefits$ (Figure 4.7). M_{min} is the minimum amount of memory that is required to execute the plan. Initially we select the minimum scheme for every operator and reserve M_{min} amount of memory, this guarantees the execution of the plan. Memory that remains after allocating M_{min} is then divided optimally among all the operators. At every step of the algorithm, the amount of memory to be allocated is M_{avail}. The procedure $GetBestScheme$ takes as input the current available memory M_{avail} and outputs the best scheme s_{best} and its corresponding operator o_{best}. We allocate the required memory to the best scheme, i.e. $Memory(s_{best})$ to the operator o_{best} and recompute M_{avail} accordingly. We throw away s_{best} and all the schemes of o_{best} which need less memory than $Memory(s_{best})$ and recompute the benefit and size values for the other schemes.(which need more memory than $Memory(s_{best})$. If no best scheme has been returned it means that memory allocation is over and the procedure returns.

The procedure $GetBestScheme$ depending on the current available memory M_{avail}, finds out which operator scheme has the highest benefit/size ratio among all the schemes of all operators that are possible with the current available memory M_{avail}. The procedure $RemoveSchemes$ removes all the schemes of o_{best} which need less memory than $Memory(s_{best})$. It also eliminates all the schemes which are not possible with M_{avail}.

$GetBestScheme(M_{avail})$
1. Take all schemes of all operators,that satisfy $Memory \leq M_{avail}$
2. if no scheme satisfies $Memory \leq M_{avail}$
3. then $return$ no best scheme
4. else
5. Select the scheme s_{best} that has maximum benefit/size
$\forall i, 1 \leq i \leq m, \forall j, 1 \leq j \leq n_i : \frac{Benefit(s_{best})}{Size(s_{best})} \geq \frac{Benefit(s_j)}{Size(s_j)},$
$Memory(s_j) \leq M_{avail}$
6. $o_{best} = operator(s_{best})$
7. $return\ s_{best}, o_{best}$

Figure 4.6: Pseudo Code for $GetBestScheme$

$RecomputeBenefits(o_{best}, s_{best})$
{Recompute the benefit of all the schemes of the operator o_{best} which have $Memory > Memory(s_{best})$}
1. for $i = 1$ to $n_{o_{best}}$ do
2. if $Memory(s_i) > Memory(s_{best})$ then
3. $Benefit(s_i) = Benefit(s_i) - Benefit(s_{best})$
4. $Size(s_i) = Size(s_i) - Size(s_{best})$
5. end if
6. $return$

Figure 4.7: Pseudo Code for $RecomputeBenefits$.

Recomputation of Benefits

Once $GetBestScheme$ returns the best scheme, we allocate memory to the scheme and recompute M_{avail}. We observe that once the operator o_{best} gets memory $Memory(s_{best})$, the benefit and size of all the schemes of o_{best} that have higher memory requirements than s_{best} change. The new $Benefit$ and $Size$ values of those schemes will be the difference between their old values and the values for s_{best}.

Once we get the best scheme using $GetBestScheme$ and allocate its memory, s_{best} need not be the final scheme for operator o_{best}. There will be other schemes of o_{best} that might not have a better $Benefit/Size$ ratio than s_{best} but have more benefit and need more memory. After recomputation of benefits and sizes, later during the process of allocation, one of these might have the highest $Benefit/Size$ and memory to be implemented. Then $GetBestScheme$ will select that scheme and allocate the additional memory to the operator. Figure 4.8 shows an example of benefit and size recomputation.

Time and Space Complexity

The time complexity of the procedures $GetBestScheme$, $RemoveSchemes$, and $RecomputeBenefits$ are $O(nm)$, $O(nm)$, and $O(n)$ respectively where n is the number of schemes and m is the number of operators. Thus, the complexity of $MemAllocate$ is $O(nm^2)$. The space complexity is 0 since no additional values are stored.

The procedure $MemAllocate$ allocates the available memory optimally among all the operators of the plan since at every step, it selects the operator scheme that gains the most per unit memory allocation thus maximizing the benefit of the query plan. Since it allocates the minimum scheme to every operator in the beginning of the allocation, for devices with very less memory, the procedure will select the minimum schemes.

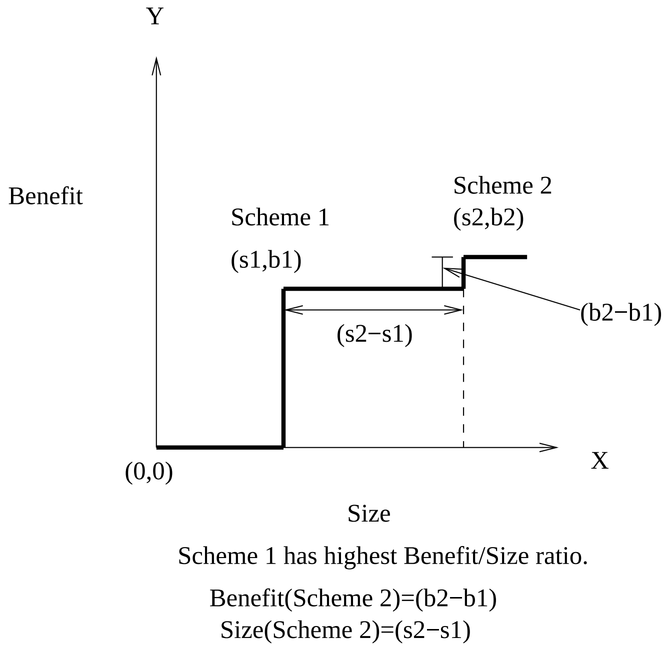

Figure 4.8: Benefit and Size Recomputation of schemes

Chapter 5

Synchronization

In many handheld applications, the main data will be available on some remote server. This information will be stored in the form of relations in a remote database and when the handheld is connected to the server, the relevant information is downloaded on the device. This activity will take place once every day or week depending on the connectivity available. Copies of a relation will be present in the remote server and the handheld device.

There are two scenarios in such information exchange between the handheld and the remote server.

5.1 Relation downloaded on the handheld

In the first case, a relation R is required to be downloaded on the handheld from the remote database. Hence, there are two copies of R, one in the main database and one in the handheld database. Both the copies are modified during the time when there is no connection and when next time the connection is done, the data is synchronized between the handheld and the remote server. We call the copy in the handheld $R_{simpold}$, copy in the remote database $R_{mainold}$, changes in the handheld ΔR_{simp}, changes in the remote database ΔR_{main}, and their synchronized copies $R_{simpold}$ and $R_{mainold}$.

We download the entire relation R to the handheld only for the first time. Subsequently, we need to transfer the differentials ΔR_{simp} and ΔR_{main} to synchronize the relation R. Suppose ΔR_{simp} and ΔR_{main} contains tuples added to the relation R, we have

$$\begin{aligned} R_{simpnew} &= R_{simpold} \quad \cup \quad \Delta R_{main} \\ R_{mainnew} &= R_{mainold} \quad \cup \quad \Delta R_{simp} \end{aligned}$$

Similarly, if the differential contain deleted tuples, we replace union $\cup$ by the difference operator $-$.

However, a problem arises when the same tuple is updated at both the handheld as well as the main server. We term this *Tuple Conflict*. *Tuple Conflict* resolution can be done by applying rules like handheld wins, Main server wins, and latest timestamp wins.

This means that we cannot allow updates to downloaded relations in the handheld directly. Instead, all those updates are recorded in a list called *Intention List*, which is a log of all the updates. We then determine whether the changes recorded in the *Intention List* can be reflected in the relation R.

5.2 Resultset downloaded on the handheld

In the second case, the individual relations need not be downloaded but we just need the result of a query at the handheld. We execute the query at the remote database and download it on the handheld. As in the first case, we execute the entire query at the remote server only for the first time. Subsequently, we compute only the difference and transfer it to the handheld. We illustrate it with a very simple example. Suppose relations R and S reside on the remote database. We do not need the relations R and S in the handheld but we need the result of the query $q = R \bowtie S$. We compute $R \bowtie S$ only once and download it on the handheld. Subsequently, we transfer only the differential. Suppose i_r is the set of tuples added to R and d_r is the set of deleted tuples. Then, we compute and transfer only $i_r \bowtie S$ and $d_r \bowtie S$ to the handheld, since

$$\begin{aligned} q_{new} = q_{old} &\quad \cup \quad (i_r \bowtie S) \\ q_{new} = q_{old} &\quad - \quad (d_r \bowtie S) \end{aligned}$$

Similarly. for each query q we have to find out what differential to compute at the remote server. We note that this resembles the incremental view maintenance problem. Hence, for any query q, we can find out the differential.

In-place updates on such a resultset in the handheld have to be propagated to the relations in the main database. This resembles the view update problem, where views are modified and we propagate the modifications to the base relations. We discuss view updation in slightly more detail in the next section.

It follows that we cannot allow updates to downloaded resultsets in the handheld. Instead, all those updates are stored in a list called *Intention View List*, which contains a log of all the updates. We then determine whether rules are violated and propagate the changes to the base relations.

5.3 View Updates

The problem of updating a database through a view is a well known problem. Update specified on a view have to be translated into updates on the underlying base relations. The updated database state then induces a new view state, and it is necessary that the new view state correspond to performing the user-specified update directly on the original view state. The following aspects need to be considered

1. Only legal view updates

   Only those view updates have to be accepted that are legal, that is, that do not introduce inconsistency to the view state.

2. Only legal database updates

   As a result of translation of a view update, a tuple has to be inserted in a base relation that conflicts with some other tuple. In such situations, the update have to be rejected.

3. Only underlying tuples affected

   Only the component tuples of those specified in the view update request should be affected by the translations.

4. Insertion of extra tuples

   When a view update is translated, extra view tuples might be introduced into the view state. If deleting these extra tuples does not introduce inconsistency, the update is valid.

5. Modification of other view tuples

   We should avoid modification of view tuples, other than those specified in the updates.

One approach was the constant complement method, but [Lan90] proves that the constraints used in this method are too restrictive and they are applicable to only a small class of views. [Kel85] provides algorithms for translating insertions, deletions, and modifications. However, the restrictions are that relations should be in Boyce-Codd Normal Form and the joins should have an inclusion dependency in them. Moreover, it does not handle nulls. [Lan90] suggest translations based on the properties of extension joins. Their approach works for all normal forms and it also handles nulls. However, functional dependencies are the only integrity constraints that have been taken into consideration.

The base relations corresponding to the views stored in the handheld database belong to a remote database and hence, we cannot assume any normal form for them. Also, most of the databases allow nulls and joins may not have an inclusion dependency. This precludes the use of techniques suggested in [Kel85] even though they are very simple to interpret. The translations suggested in [Lan90] do not have any restriction and can be used as the basis of our synchronization engine. Most of the techniques need data and metadata about the base relations. Hence, it is not possible to decide whether the updates recorded in the *Intention View List* are valid without getting connected to the main database.

5.4 Logs from remote server

In order to reduce the data flowing between the handheld and remote server, we argued that the data synchronization problem is identical to incremental view maintenance and view update problem. For any query q the result of which is needed in a handheld, we need to keep track of the changes to the input relations and use the rules of view maintenance to compute the differentials that need to be computed.

The changes to any relation in a database are recorded in system logs. However, these logs are internal to the database system and not available for viewing to a database user. Hence, we cannot expect to get access to these logs.

5.4.1 A Middle-Tier Approach

One way to store the log information is to use the interface which is responsible for all the updates done in the main database. This middle layer will now keep track of all the changes to the relations in its own logs and periodically, the handheld can ask for the logs relevant to it. All the updates that are done on the handheld, also are propagated to the main database through this intermediate layer. The checks that need to be done when view updates in handhelds have to be reflected in the main server, can be implemented in this middle layer. Problem with this approach is that the modifications to the middle-tier interface might not be permitted by the administrator.

5.4.2 Triggers

Another way to do the logging manually is to write triggers at the main database. A trigger is a statement that the database system executes automatically as a side effect to a modification to the database. We can create triggers for the updates on the relations. These triggers will log all the changes into some other log relation which the handheld can use.

5.4.3 Audit Trail

Most of the database systems have a mechanism of performing an audit. The audit information can be used to solve the problem of data synchronization. The auditing of the updations can be activated and periodically, the audit information can be sent to the handheld. The audit will contain the update statements that have been applied to the relations. Once we download the audit in the handheld, we should execute those statements in the handheld database on the downloaded data.

Though either of the above solutions work in case of view maintenance problem, for propagation of view updates we need to have a middle tier which checks whether the view updates are legal and if yes, generate the appropriate modifications for the main database.

Chapter 6

Transaction Management

In order to ensure the integrity of the data, the DBMS must maintain the ACID properties of the transactions. Atomicity will ensure that the set of actions executed on behalf of a transaction follows the *all or nothing* rule. The integrity constraints defined on the handheld data can be handled by standard integrity constraint management schemes.

Handhelds like Simputer also act as community digital assistant and smartcards are used as personalizing agents. When some transaction has to update data in the handheld database as well as the smartcard e.g. at the end of the day the farmer is paid his profit and same amount is deducted from the total, we have to ensure that both the changes (profit entry in the smartcard and deduction from the total) take place or none. Hence, some form of global atomicity has to be maintained. A similar situation arises when the handheld participates in distributed transactions with the main database.

The handheld is primarily a single user machine. So, one might question the need of concurrency control in the handheld database. There might be situations when the handheld is connected to some main DBMS for data exchange. Then, the local user may be executing some transaction and some other transaction may be running on behalf of data synchronization. Even when one user is currently using the system, there can be some long transactions doing some aggregation while the user might be updating his/her guestbook. Hence, we need a small amount of concurrency.

We discuss some of the issues related to concurrency control and commit processing.

6.1 Concurrency Control

As access to memory is faster than disk access, we can expect most of the transactions to complete much faster than disk-based systems. Hence, if we use lock-based concurrency control, locks would not be held for long durations which suggests that lock contention may not be important. Disk-based systems have small locking granules like records to reduce contention. If lock contention is already less, the advantages of small granules are lost. Hence, we can go for relatively large locking granules. If we choose the entire database as the locking granule, then it amounts to serial execution of transactions. Concurrency issues for serial execution are almost eliminated. However, even most of the transactions are short, we do need some form of concurrency if we want to execute them with the long running ones.

In a recent work, Korth et al. [BK02] propose serialization of transactions in main memory databases. In the application domain that they consider, most of the transactions are short and lock contention is extremely low, which is similar to our system. The main aim is to improve response time rather than throughput. Serial execution of transactions is ensured by using a mutex on the entire database. All the overheads associated with locking as well as deadlocks are eliminated. By using timestamps and additional mutexes, it allows transaction processing during flushing of log records and at the same time ensures that no transaction commits on the state of having read uncommitted data.

Typically, locks are implemented by some hash table that stores the information of the locked objects. Sometimes, a small number of bits are used to represent the lock status of an object. One bit can be reserved to denote whether the object is locked or not and another bit to signify whether there are transactions waiting to lock it. However, the identity of waiting transactions are stored in a table. Though there is an extra space overhead, the hash table lookup is avoided in cases like a transaction locks a free object, updates it, and releases the lock before any other transaction waits for it.

6.2 Commit Processing

Commit processing and recovery can be done either by shadowing or log-based schemes.

1. Shadow updates

   The two main drawbacks of shadow based updates which makes it unacceptable for disk-based systems are (i) Data locality is lost and (ii) Concurrent transactions are difficult to handle. However, data being in memory, data locality is not an issue. But shadow updates may not be a good option with respect to our storage model which is pointer-based. Even if one domain value is changed, its location also changes which means all tuple pointers need to be updated. Also, size of the shadow is an issue since we are constrained by less memory and stable storage.

2. Log-based update

   Log-based updates are better suited for our storage model since it accommodates pointer-based models and also no shadow copies are maintained. However, we have to bear the additional cost of maintaining the logs. Techniques like Write-Ahead Logging have to be employed. One optimization to traditional logging is to do pointer-based logging rather than value-based logging. We log the pointers to the values instead of the values itself. For the domain values we log the domain pointer and for non-domain values the tuple pointer. Such a scheme would reduce the storage cost of the log.

Since we have very limited memory, our buffer replacement strategy can only be *Steal*. If we output the dirty blocks to the stable storage of the handheld, then undo is required. However, we observe that if these dirty blocks can be written on some other stable storage, we need not undo the transaction. This alternate form of stable storage is available in the smartcards. If the smartcard is connected to the handheld, we can use it to store the dirty blocks. We can provide a lookup table where for every uncommitted transaction, we store the information whether its dirty blocks have been output to the handheld or the smartcard. If all the dirty blocks of a transaction are output to the smartcard, no undo is needed for it. As most of the transactions do not have many dirty blocks, these blocks are likely to fit in smartcard stable storage. A more detailed literature survey of the existing techniques for concurrency control and recovery is left as a future work.

Chapter 7

The Database System

The system is being developed using the C programming language. The codebase is distributed over several subdirectories which form a tree structure (Figure 7.1). The backend database code is present in *simpdb/src/backend* and the include files in *simpdb/src/include*. Recursive makefiles are used in every subdirectory to build the system. The tools lex and bison have been used to write the SQL parser.

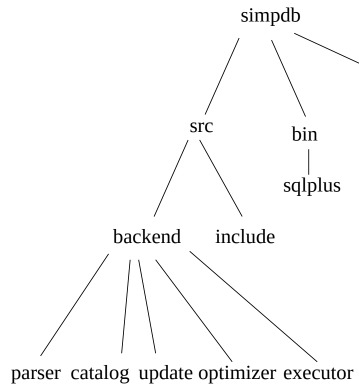

Figure 7.1: Codebase directory structure

7.1 Storage Manager

The storage manager code is distributed in two directories, *catalog* and *update*. The *catalog* directory contains the functions that deal with the storage manager interface with the system catalog. Every storage manager has its own system catalog. The *update* directory implements the functions which reads and writes tuples from the database. Every storage model implements the following functions for storing and retrieving data. The following the functions for ID Storage.

id_FILE *id_open(RelationName);
int id_fseek(id_FILE *fp, long numberoftuples, int whence);
int id_fread(void *buffer, int tuple_size, int count, id_FILE *fp);
int id_close(id_FILE *fp);

void InsertIdValues(RelationInfo, AttributeValueInfo);

All the storage models have their own *catalog* and *update* directories. The storage models Flat Storage, Domain Storage, Ring Storage, and ID Storage have been implemented presently. In addition, ID based Join Index model has also been implemented. Bitmap Storage model is being implemented.

The data types that are being currently supported are INTEGER, FLOAT, AND CHAR. CREATE TABLE and INSERT statements are fully supported with the above data types.

7.2 Query Optimizer

The query optimization code is present in the directory *optimizer*. The optimal operator order is found out using a dynamic programming algorithm. This is being used to determine the query execution plan. The code for operator order determination is available in *bestplan.c*. We have implemented the exact allocation algorithm as well as the heuristic allocation algorithm. The code for memory allocation is present in *memalloc.c*.

7.3 Query Execution

The query execution code is present in the directory *executor*. The operators *select* and *project* have been implemented fully. *Nested Loop Join*, *Indexed Nested Loop Join*, and *Hash Join* have been implemented. All the operators have been implemented using the iterator model. Among the aggregate operators, the COUNT operator has been implemented using the *nested loop* and the *buffered* approach. The rest of the aggregate operators, AVG, MIN, MAX, SUM are being implemented. All types of SPJ queries are supported alongwith the COUNT operator.

7.4 Building the system and Sqlplus

To build the system, go to the directory *simpdb* and run the makefile present there. The recursive makefiles inside the directories are executed in turn and finally the SQL parser is build in the directory *simpdb/bin*. The SQL parser takes SQL statements from command line as well as files and executes them.

Chapter 8

Performance Analysis

Performance analysis of a database in the handheld environment is quite complex. Parameters like processor speed, caching strategy, RAM and Flash RAM speed will strongly affect the response time of a query in a handheld device. [BBPV00] measure the performance of their query engine for smartcards on a Pentium-486 computer and varying the system parameters. [BBPV00] measure the performance on a calibrated platform. However at the end of the day, the database will be running in a lightweight device. To determine the efficiency of the database techniques accurately, we feel that the database should be ported on a handheld computing device and queries executed on the device. Only then we expect to get real performance numbers.

8.1 Experimental setup

We planned to port the database system on the Simputer[Sim]. It is a low cost handheld device produced in India. . Our version of the Simputer runs on a Intel Strong Arm processor at 200MHz. It has 28MB of Flash RAM for permanent storage and 24 MB of DRAM. The gcc cross compilation toolkit for Strong Arm processor was used to cross compile the code for the Simputer. Simputer runs on Linux and has other computationally intensive applications running on it.

For performance testing, we need schema, data, and queries which suit the needs of a handheld device. Hence, we select the healthcare schema and queries used in [BBPV00] and described in Figure 4.1. The dataset taken was a relatively large one (Doctor(91), Drug(77), Prescription(2155) , Visit(830)). The datasets used are the same as [BBPV00]. We executed all the 4 queries on the Simputer using the dataset. We choose the query Q1 as the representative for complex joins and query Q4 for aggregate queries. Query Q1 has 2 selections and 3 joins while query Q2 performs an aggregation over two attributes. We measure the absolute response time of each query, varying the amount of memory available for execution. We stored the data in two different storage schemes, Flat Storage and ID Storage. We optimize both the queries using the exact memory allocation method as the heuristic memory allocation method. Figure 8.1 and Figure 8.2 show the results for query Q1 and query Q4.

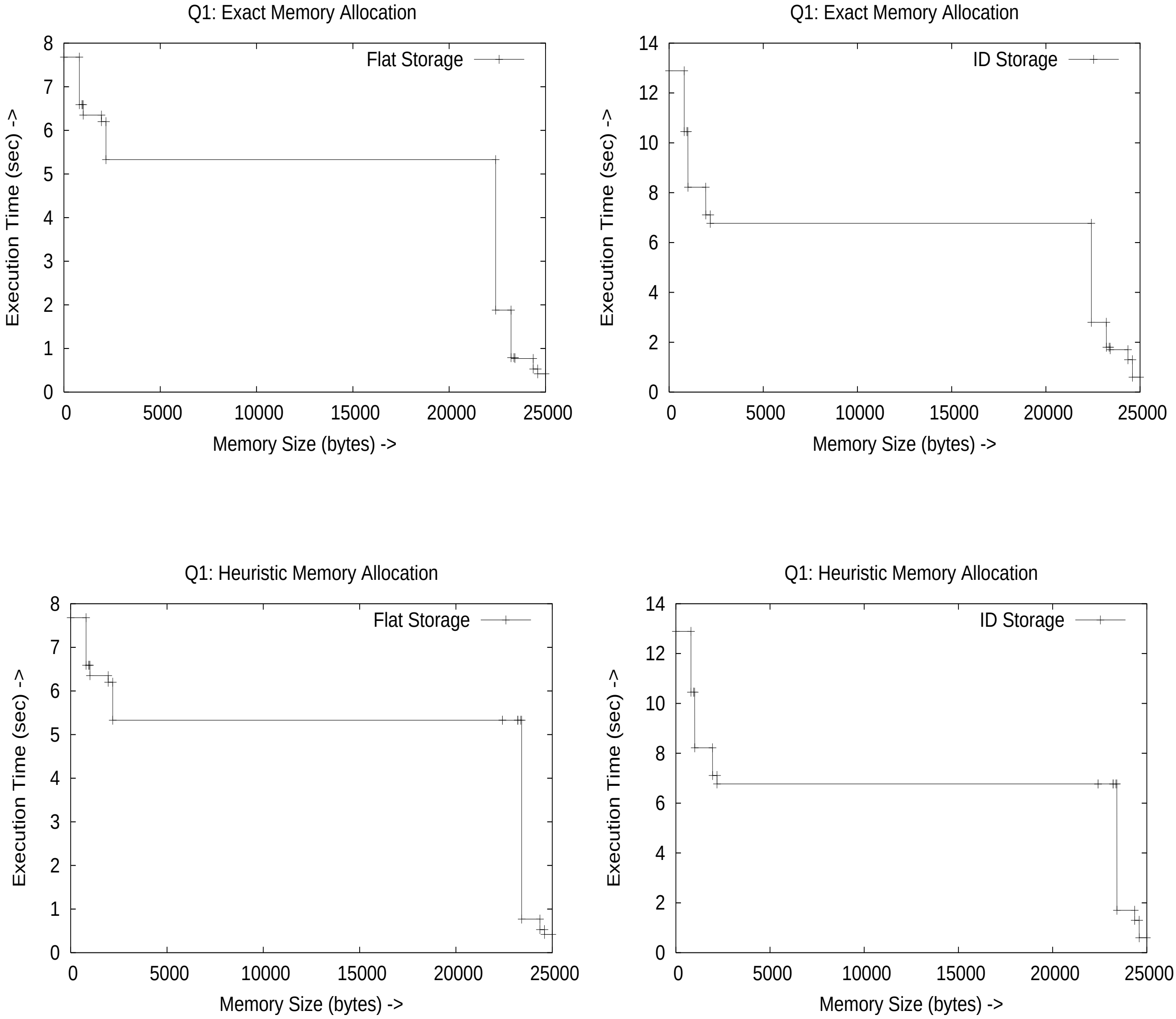

Figure 8.1: Performance Results for Query Q1

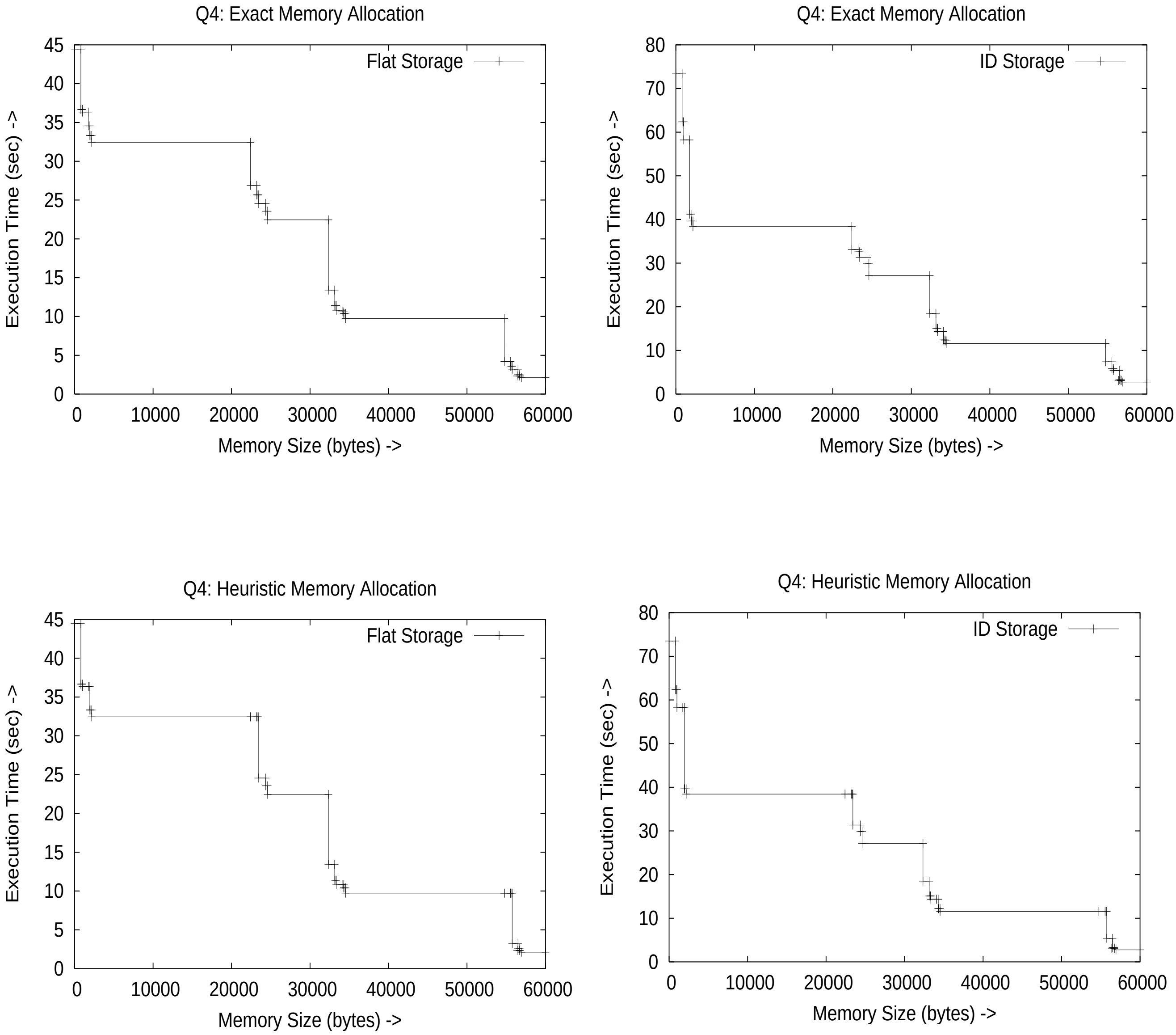

Figure 8.2: Performance Results for Query Q4

8.2 Observations

The response time as expected was the highest with the query plan for zero memory where the operator schemes chosen by the optimizer were all nested loop schemes. The time was least when least cost operator schemes were possible and indices were constructed and aggregate values stored. In all the graphs, the query plans change at finite number of points. These points are the memory division points. The query plans generated remain the same till the optimizer finds that memory is available for an operator scheme that leads to a better plan. It would be unfair to compare our numbers with those in [BBPV00] for two important reasons (i) [BBPV00] executed the queries in a Pentium-486 system while we executed them on the Simputer. (ii) Their optimizations were specific to the smartcard platform and domain-based storage while our query engine is meant for all types of devices and all storage schemes.

The response times for query Q1 ranged from 7.6 s to 0.4 s for Flat Storage and 12.8 to 0.6 for ID Storage. This was expected considering the complexity of the joins and the computational power of the handheld device. The join order determined in phase-1 was QEP 2 of Figure 4.2. We number each operator from the leaf of the plan tree. Operator 2 had the highest input tuples and was the most costly. Our heuristic memory allocation algorithm generated the same query execution plan as the exact allocation method except for a few points. The points where it differed from the exact solution were those in which the scheme having highest benefit was not the one having the highest benefit/size. As we increased the memory, better schemes were selected for every operator which resulted in better query plans and the response time reduced. We observe that for a long memory interval (2.1K to 22.4K), the query plan remain the same. This was because 22.4K was needed to build index for operator 2 and the best schemes for the other operators were possible with 2.1K. At 22.4K, the scheme for operator 2 changed from nested loop to hash join which resulted in a substantial reduction in time. The other operator schemes changed to nested loop. With a memory of 24.5K, the least cost schemes were selected for all the operators and the response time dropped to 0.4 s. The response times for ID Storage were more than Flat Storage. This was expected because of the extra cost of projecting out a tuple in ID Storage. The time difference was more when nested loop join was selected for operator 1, since the input relation to it *Doctor* has a domain on $Name$.

For aggregate query Q4 the response times ranged from 44.4 s to 2.1 s for Flat Storage and 73.5 s to 2.76 s for ID Storage. Q4 had groupby on two attributes , hence such a response time was not unexpected. The join order determined in phase-1 was QEP 2 of Figure 4.3. The groupby operator at the root was the most costly operator and influenced the execution time the most. Operator 2 also affected the execution time but not as pronounced as the groupby operator. The heuristic algorithm generated the same query plan as the exact algorithm in this case also barring a few points. With increasing memory, query plans became less costly. For three intervals, 2.1K to 22.4K, 24.5K to 32.4K, and 34.5K to 54.7K the query plan remained the same. The first interval is the same as query Q1 where the scheme for operator 2 affected the time. With 24.5K, the best schemes were selected for all the joins but for the groupby operator the scheme possible was nested loop. With 32.4K, groupby scheme changed to Buffered Aggregation and the execution time dropped to 13.4 s from 22.5 s. However, schemes for the joins changed to nested loop. The interval from 34.5K to 54.7K is similar to the first interval 2.1K to 22.K with the difference that the scheme for groupby had changed. With 56.9K, the response time was least. The numbers for ID Storage again followed the same pattern as in Q1, the difference with Flat Storage being more when nested loop join was the scheme for operator 1.

Our performance results show that the query execution plan chosen at any stage was the best possible plan with the available memory. As we increased the memory for query execution, the response time reduced. Our heuristic memory allocation generated the same plan as the exact method for most of the memory points. The reason for this is that even after selecting a scheme for an operator, we do not throw away the other schemes and recompute their benefits. For devices which cannot afford the exact algorithm, our heuristic algorithm will work fine for most of the cases. The response time for aggregate is quite high with minimum memory. However, for devices with minimum memory, we expect the size of the datasets to be much smaller than ours. Our query engine is able to exploit efficiently any additional memory that is available for query execution. It solves the problem of optimal memory allocation among the database operators in a handheld database. It is evident that query optimization is a must for complex joins and aggregate queries.

Chapter 9

Summary and Future work

The Storage manager, query optimizer, and query executor has been implemented and the database code ported to the Simputer. The system now supports SPJ queries and aggregate COUNT queries. We created several datasets in the Simputer and ran sample queries to verify the system.

We proposed two new storage models, ID based Storage and Bitmap based Storage. We proved that ID based Storage outperforms Domain based Storage for most of the data that would reside on a handheld. Under certain conditions, we showed that Bitmap storage scores over Ring Storage. All the storage models were examined based on their storage cost and some performance factors. We presented a table which would serve as a guideline for a storage manager to decide which storage model to choose for an attribute. We also highlighted the need of allocating memory optimally amongst the operators in a plan. Several schemes for evaluating the database operators were proposed, each of them having different cost and using different amount of memory. We proposed an exact and heuristic solution to the memory allocation problem. Our solution was based on the 2-phase approach. The performance evaluation shows that our query engine generates the best query plan depending on the amount of memory available in the handheld device.

Ongoing and Future work

As part of ongoing work, we are developing a synchronization utility which will download data from a remote server and store it in the handheld database. The utility will also determine the best storage scheme for the relations by analyzing the metadata. If there are any queries to be posed on the data, it will compute the query execution plan and store it in the handheld device.

The query optimizer in its current form is a 2-phase optimizer. However, the operator order decided in phase-1 might itself be suboptimal for the available memory. 1-phase optimization is the ideal way to optimize a query. As part of future work, we plan to do a study of the issues involved in 1-phase approach to the memory allocation problem and examine the feasibility of developing a 1-phase optimizer.

Resources are not uniform across handheld devices. A PDA has more computing power than a mobile phone, hence a PDA can support more DBMS features than a mobile phone. We plan to build a DBMS module toolkit; a library of DBMS modules. Modules from this toolkit can be plugged into a system depending on type of the application and computing power of the device.

Handheld devices have limited display and the users may wish to view only the first-k results of a query. [CK97] proposes a new operator for this and methods to implement it. We wish to examine the feasibility of such an operator in our system.

Handhelds have limited functionality and most of the users use it for manipulating a specific set of data. A database system which caters to only specific applications need not have an SQL interpreter. Instead the frontend can directly call some modules at the backend to perform the queries. These backend modules will be similar to stored procedures in existing database systems. We wish to explore more on this front.

References

[AHK85] A. Ammann, M. Hanrahan, and R. Krishnamurthy. Design of a Memory Resident DBMS. In *IEEE COMPCON*, 1985.

[BBPV00] C. Bobineau, L. Bouganim, P. Pucheral, and P. Valduriez. PicoDBMS: Scaling down Database Techniques for the Smartcard. In *VLDB*, 2000.

[BK02] Stephen Blott and Henry F. Korth. An Almost Serial Protocol for Transaction Execution in Main Memory Database Systems. In *VLDB*, 2002.

[BPA03] L. Bouganim, P. Pucheral, and N. Anciaux. Memory Requirements for Query Execution in Highly Constrained Devices. In *VLDB*, 2003.

[CK97] Michael Carey and Donald Kossmann. On Saying Enough Already in SQL. In *ACM SIGMOD*, 1997.

[DKO+84] David J DeWitt, Randy H Katz, Frank Olken, Leonard D Shapiro, Michael R Stonebraker, and David Wood. Implementation Techniques for Main Memory Database Systems. In *ACM SIGMOD*, 1984.

[DVRM99] Anindya Datta, Debra VanderMeer, Krithi Ramamritham, and Bongki Moon. Applying Parallel Processing Techniques in Data Warehousing and OLAP. In *VLDB*, 1999.

[GH90] Le Gruenwald and Margaret H.Eich. Choosing the Best Storage Technique for a Main Memory Database System. In *IEEE Conference on Information Technology, Jerusalem*, 1990.

[Gig01] E. Giguere. Mobile Data Management: Challenges of Wireless and Offline Data Access. In *ICDE*, 2001.

[Gra93] G. Graefe. Query evaluation techniques for large databases. In *ACM Computing Survey*, 1993.

[HSS00] A. Hulgeri, S. Sudarshan, and S. Seshadri. Memory Cognizant Query Optimization. In *COMMAD*, 2000.

[Kel85] Arthur M. Keller. Algorithms for Translating View Updates to Database Updates for Views Involving Selections, Projections and Joins. In *ACM PODS*, 1985.

[KFW+03] M. Kersten, M. Franklin, G. Weikum, D. Keim, A. Buchmann, and S. Chaudhuri. A Database Striptease or How to Manage Your Personal Database. A Panel Discussion. In *VLDB*, 2003.

[KLLP01] J.S. Karlsson, A. Lal, C. Leung, and T. Pham. IBM DB2 Everyplace: A Small Footprint Relational Database System. In *ICDE*, 2001.

[Lan90] Rom Langerak. View Updates in Relational Databases with an Independent Scheme. In *ACM PODS*, 1990.

[LC86] T. Lehmann and M. Carey. A Study of Index Structures for Main Memory DBMS. In *VLDB*, 1986.

[MFHH02] S. Madden, M. J. Franklin, J. Hellerstein, and W. Hong. TAG: a Tiny AGgregation Service for Ad-Hoc Sensor Networks. In *Intl. Conf. on Operating Systems and Implementation*, 2002.

[MS83] M. Missikov and M. Scholl. Relational Queries in a Domain Based DBMS. In *ACM SIGMOD*, 1983.

[OLi] Oracle Corporation, Oracle 9i Lite, Oracle Documentation.

[PVJ90] P. Pucheral, P. Valduriez, and J.M.Thevenin. Efficient Main Memory Data Management using the DBGraph Storage Model. In *VLDB*, 1990.

[RR99] Jun Rao and Kenneth A. Ross. Cache Conscious Indexing for Decision-Support in Main Memory. In *VLDB*, 1999.

[SACL79] P. Selinger, M. Astrahan, D. Chamberlin, and R. Lorie. Access path selection in a relational database management system. In *ACM SIGMOD*, 1979.

[Ses99] Praveen Seshadri. Honey I Shrunk the DBMS. In *ACM SIGMOD*, 1999.

[SG00] Praveen Seshadri and Phir Garrett. SQL Server for Windows CE - A Database Engine for Mobile and Embedded Platforms. In *ICDE*, 2000.

[Sim] The Simputer. http://www.simputer.org.

[Sma] Small Databases are Beautiful, Database Trends and Applications, August 2003. http://www.dbta.com.

[SSK02] S. Sudarshan, A. Silberschatz, and H. Korth. *Database System Concepts*. McGraw-Hill International Fourth Edition, 2002.