Efficient Data Management on Lightweight Computing Devices

Rajkumar Sen and Krithi Ramamritham
Indian Institute of Technology, Bombay
Mumbai-400016, India
{rajkumar, krithi}@cse.iitb.ac.in

Abstract

Lightweight computing devices are becoming ubiquitous and an increasing number of applications are being developed for these devices. Many applications deal with a significant amount of data and involve complex joins and aggregate operations which necessitate a local database management system on the device. However, scaling down the database has not been easy as these devices are constrained by limited stable storage and main memory. Even though resources in handhelds are increasing with every new device, optimum utilization of these limited resources is a must for such a database system. Storage models that lead to compact data storage need to be designed. Query plan generation is influenced by the amount of memory available and the underlying storage model. The limited memory needs to be optimally allocated among the database operators. We propose a new data storage model, ID Based Storage which reduces storage costs considerably. We present an exact and a heuristic method for allocating memory among the database operators and compare both the methods. Our query processing strategy guarantees the best query execution plan for a given handheld device.

1. Introduction

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 deal with a significant amount of data leading to the need of embedded database support on these devices [3]. The queries go beyond simple SPJ queries but still have to be locally computed on the device [6]. 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 [15]. Applications for PDAs execute complicated join and aggregate queries on the device resident data. Thus, there is an increasing need to facilitate execution of complex queries locally on a variety of lightweight computing devices[14, 20].

However, scaling down the database techniques poses challenges since these lightweight devices come with very limited computing resources. The amount of main memory and stable storage available in such devices is relatively small. Also, the devices are not uniformly endowed with resources, the computing capability and main memory of a cellphone differs from that of a PDA. It is essential that the available resources be utilized optimally for a database system that is developed for such devices.

The already limited stable storage has to accommodate the operating system as well as the database system code, which means even 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. Different storage models have different storage and update costs. The selection of the best storage model for an attribute depends on the size of the relation, selectivity and length of the attribute, frequency of updates, and the nature of queries.

As far as the choice of query processing techniques are concerned, RAM is the most critical resource in these devices. Existing approaches [2, 10, 6, 7, 13, 21] have used minimum memory algorithms for every operator. This can lead to poor performance for complex queries involving several joins and aggregates. Query execution time for complex queries can be reduced by using operator algorithms which build in-memory indices and save the aggregate values. Query plans should be generated in such a way that they make effective use of the available memory. Optimal memory allocation among the database operators is a must if we need to ensure the best usage of memory. Another factor that influences query execution is the storage model used to store the relations. Each storage scheme entails a different cost for selecting and projecting a tuple, hence the cost of a query execution plan will vary across storage schemes. Thus, the selection of a query execution plan for a handheld device should be governed by (i) the amount of memory available, and (ii) the underlying storage model. Memory and storage model cognizant query plan generation is hence essential. Also, the query optimizer itself should not be too complex to be executed on the handheld device.

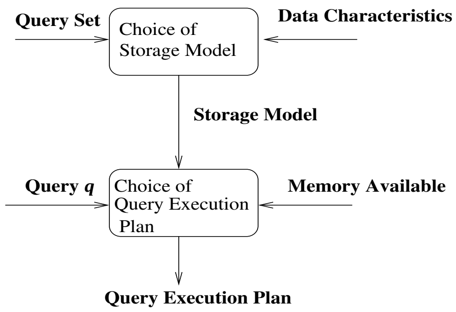

Figure 1. Choice of storage scheme and query execution plan

Figure 1 characterizes the requirements of the database system. The system should select the storage model based on data characteristics and the expected type of queries. For any query q, depending on the storage model and amount of available memory, the optimal query processing technique should be determined.

Lightweight versions of some of the popular DBMSs like Oracle Lite [2], Sybase Adaptive Server Anywhere [10], IBM DB2 EveryPlace [13], and Microsoft SQL Server for Windows CE [21] have been developed with a reduced code footprint by stripping down database features. IBM DB2 EveryPlace[13] organizes data in a flat way; records are stored sequentially and column values are stored contiguously in the record. The query optimizer does not generate and examine different evaluation plans for a query. Only nested loop algorithm is used to evaluate joins.

Instead of having a statically defined storage scheme and query processing technique independent of the database application and the device resources, we need to have a dynamic approach where we choose the storage model and query processing schemes depending on the application characteristics and the resources present in the device. There has been some work to this effect. For example, [6] proposes a new storage model called Ring Storage for smartcards 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 optimizations suggested for aggregate queries rely on the underlying storage and index model.

[7] proposes a query execution model that has a lower bound in terms of RAM usage. The operator 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 measure the cost of an operator in terms of the number of tuples accessed and distribute the memory among the operators using a heuristic cost model (The memory allocation algorithm is not mentioned in [7]). They provide guidelines to estimate the RAM resource of a hardware platform according to the requirements. Their performance study mentions 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, adding indices and saving aggregate values are the effective alternatives to speed up query processing.

The non-uniformity of resources across handheld devices can result in a need for a special purpose database system for each type of device, e.g. smartcards, cellphones, PDAs. An approach that can cater to the needs of different types of lightweight computing devices is needed. Our contributions lie in the development of such an approach. Our contributions are two-fold:

1. In order to reduce the storage cost new storage techniques may be needed and the best storage scheme should be selected based on data characteristics and nature of queries.

2. The query processing engine should optimally allocate the available memory among the operators. It should choose the minimum cost query plan for a given handheld device depending on the amount of available memory and the underlying storage scheme.

In this paper, we address these issues through the following contributions:

- We utilize a novel storage model, ID based Storage which is an improvement over the existing Domain Storage Model[5]. For most of the relations that would reside on a small device, ID Storage wins over Domain Storage and thus reduces storage costs considerably. It also provides a unidirectional Join Index between a foreign key-primary key pair thus speeding up joins. We examine the suitability of this scheme vis-a-vis existing storage schemes and depending on the data and query characteristics, select the best storage scheme for an attribute.

- Given the need for optimal allocation of memory among the database operators based on the cost function of the operator algorithms, we show that the exact memory allocation algorithm proposed in [12] can be modified to be used in our context. However since the time and space complexity of the exact algorithm can prevent it from being used in some devices, we also propose a heuristic solution with a reduced complexity based on the benefit of an operator per unit memory allocation.

We illustrate the efficiency of our storage management and query processing techniques through a performance evaluation. We have implemented our approaches on the Simputer [4], a handheld device, and performed our experiments on the device. Our experience with this implementation indicates that (a) ID based Storage can lead to lower storage cost for the dataset used in [6] and (b) Our query processing techniques based on the memory allocation algorithms always select the query plan with minimum cost and so, memory and storage model cognizant query optimization is both feasible and essential.

The rest of the paper is organized as follows. Section 2 covers the storage management issues and introduces ID based Storage. Section 3 highlights the need for query optimization for complex queries and presents the exact memory allocation and our heuristic memory allocation algorithm. In Section 4, we do a performance analysis of our storage manager and query engine and we conclude in Section 5.

2. Storage Management

Lightweight devices are constrained by very limited stable storage. Hence, storage models should lead to compactness in the representation of data and indices. In this section, we quickly review some compact storage models and then introduce a novel approach that can often lead to further compactness.

The simplest way to store data is *Flat Storage(FS)* where tuples are stored sequentially. However, it consumes a lot of space since it stores duplicate values. Also, due to the absence of index structures, tuples have to be accessed sequentially, increasing query processing times.

[5] introduced the concept of pointer based models proposing a new storage model called *Domain Storage(DS)* where duplicate values are eliminated by partitioning the attribute values into domains which are sets of unique values. Tuples reference the values of the attributes by means of pointers. This is a compact structure since one domain can be shared among multiple attributes. However, tuple creation, update, and deletion become more complex as there might be insertion and deletion of values in the domain structure.

[6] propose a storage model called *Ring Storage(RS)*, a modification of Domain Storage that addresses index compactness. It stores value-to-tuple pointers in the domain structure and tuple-to-tuple pointers that connect two tuples that have the same value for the domain attribute. An index structure in the form of a ring is formed from the domain values to the tuples. The index storage cost reduces to a minimum, the price paid is the extra cost in projecting a tuple to the ring attribute. A bidirectional Join Index is formed between a primary key-foreign key pair which can be used to speed up query processing.

To compare the storage cost associated with each of the storage models. 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. $s = \frac{D}{N}$

Cost(Flat Storage)=$N * L$
Cost(Domain Storage)=$N * p + D * L$
Cost(Ring Storage)=$N * p + D * (L + p)$

The cost of Flat Storage and Domain Storage are comparable when $S = \frac{L-p}{L}$. 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. Domain Storage is the clear winner for higher L and lower S values.

2.1. ID Based Storage

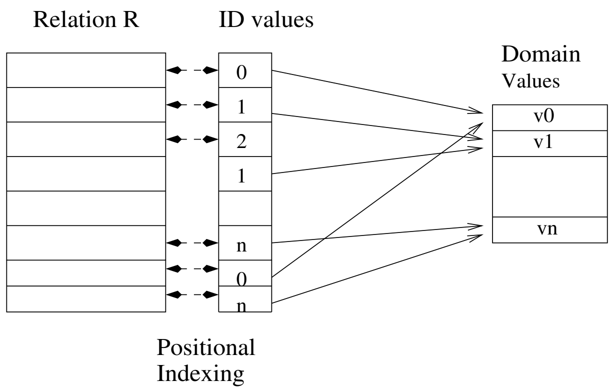

Figure 2. ID Based Storage Model

In Domain Storage, a pointer of size p, typically 4 bytes, is used 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 distinct 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 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. Inspired by recent work on cache conscious indexing [16] and on Data Index [8] we could use an integer to represent the IDs. However, 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 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^{8l} where l is the length of the identifier, then l needs to be increased. In this way, we increase the length of the identifiers only when it is absolutely required. 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. Figure 2 illustrates our storage model. Note that the solid line portrayed pointers shown in the figure are not actually stored, they are only to indicate references in the figure.

The identifiers to the domain values are assigned in increasing order starting from 0 (We start from 0 in order to use the ID value as an offset into the domain structure) 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 (since we start assigning ID values from 0) than 2^{8l}, then the length of 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.

Efficient 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 with 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 the 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 $\frac{1}{S}$.

Do we really need to delete a domain value that is not being currently referenced?. The fact that a domain structure has been created for that attribute means a future insertion might reference that domain value. It does makes sense not to delete an unreferenced domain value. However, we need to keep a count of such "holes". If the number of holes exceed a *deletion threshold*, we perform deletion.

Avoiding the 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. When the length of the IDs increases, we increase the *deletion threshold* thus suspending deletion of domain values at the boundaries.

Projection Index

Since the length of the identifiers grows and shrinks 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 store separately, with each identifier being in the same position as its corresponding base relation. This property also called *Positional Indexing* [8] 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.

2.1.1 Comparison of Storage Costs

We compare ID Storage model with Domain Storage and examine the conditions under which ID Storage saves more space than Domain Storage. ID Storage is only a data storage model without any selection indices. Hence, we cannot compare it with Ring Storage.

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

The cost for the two models becomes equal when $p = \lceil \frac{\log_2 D}{8} \rceil$. 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. This is likely to be the case for applications running on small devices since the cardinality of the relations will not be very 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.

2.2. ID Based Primary Key-Foreign Key Join Index

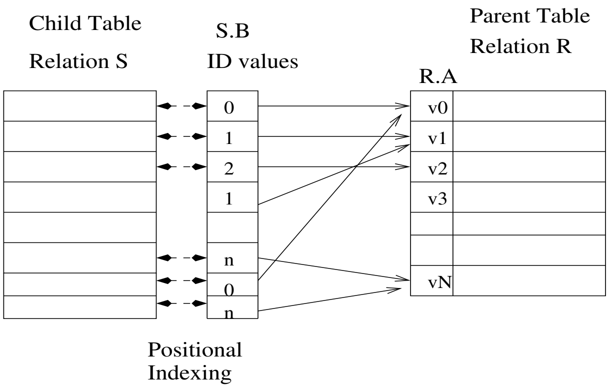

Figure 3. Join Index for Primary key-Foreign key

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 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 the *Join Index* of [8].

ID Based Join Index vs. Ring Storage Join Index
Ring Storage also provides a bidirectional Join Index between a primary key-foreign key pair to speed up join between parent table and child table. Hence, we need to compare ID Based Join Index with the Ring Storage Join Index. Suppose, N and D are the cardinalities of the child table and the parent table respectively.

Cost(ID Based Join Index) = $N * \lceil \frac{\log_2 D}{8} \rceil + D * L$
Cost(Ring Storage Join Index)= $N * p + D * (L + p)$

Ring Storage has a greater storage cost than Domain Storage. In Section 2.1.1, we already showed that for applications running on lightweight devices, ID based scheme wins over Domain Storage. Hence, ID based Join Index outperforms Ring Storage in terms of storage cost.

Storage Model	Good When
Flat Storage	$S > \frac{L-p}{L}$, Frequent Updates
Domain Storage	$S < \frac{L-p}{L}$, $p < \lceil \frac{\log_2 D}{8} \rceil$, Few Updates
ID Storage	$S < \frac{L-p}{L}$, $p > \lceil \frac{\log_2 D}{8} \rceil$, Few Updates
Ring Storage	$S < \frac{L-p}{L}$, Few Updates Frequent selections on the attribute
ID Join Index	$S < \frac{L-p}{L}$, $p > \lceil \frac{\log_2 D}{8} \rceil$, Few Updates Primary key-Foreign key relationship

Figure 4. Choosing the best storage scheme

To summarize, ID based Storage has the following advantages

- Instead of storing a static pointer, the model uses extendable IDs. A lot of space is saved especially when the number of domain values is less, which will be usually the case for small devices.
- Using the same scheme, we can represent primary key-foreign key relationships and get a join index almost for free. This leads to efficient join of the parent and child tables.

2.3. Choosing the best storage model

Figure 4 summarizes the conditions in which a storage model would perform well. These serve as guidelines to decide which storage model to be used for an attribute.

3. 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 read costs are equivalent to main memory reads, so there is no need for 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.

3.1. Features desirable in Query Optimization for lightweight computing devices

3.1.1 Need for Left Deep 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 [18] 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 base 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.

3.1.2 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[12]. This problem is crucial since handhelds are constrained have limited 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, nested loop 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. Since the amount of memory available in handhelds is increasing with every new device, 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. Thus, optimal memory allocation among the operators becomes necessary in a handheld device.

1-Phase and 2-Phase Memory Allocation

There are two approaches to solve the memory allocation problem: a 1-phase and a 2-phase approach [12]. 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 plans. 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.

Model	Projection	Selection
Flat Storage	R	$N * R$
Domain Storage	$(1 + k) * R$	$N * (1 + k) * R$
ID Storage	$(1 + 2 * k) * R$	$N * (1 + 2 * k) * R$
Ring Storage	$(1 + k * (\frac{\frac{N}{D_i}}{2})) * R$	$((\frac{D_i}{2}) + (\frac{N}{D_i})) * R$

Figure 5. Projection Costs for storage models

3.1.3 Need to consider Storage Model during Query Optimization

The underlying storage model used to store the relations also influences query processing. The cost of selecting and projecting a tuple is different for each storage scheme. Suppose R is the flash memory read cost and k is the number of domain attributes. Then the cost of projection and selection in different storage schemes are as shown in Figure 5. The cost of a query execution plan depends on the selection and projection costs and hence will vary across storage schemes.

3.1.4 Need to consider characteristics of complex queries

Existing approaches to query processing in small devices [6, 7, 13, 21] 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 and aggregations over several relations. The cost of a query execution plan depends on the operator algorithms used and the ordering of the operators. Execution time for complex queries can be reduced by using non-nested loop operator algorithms that gives better performance. The operator order also plays a crucial role since it determines the size of the subtree over which the iterators reiterate.

The ordering of the operators depends on the underlying storage scheme since the cost of selection and projection depends on the storage scheme. The cost of projection mainly depends on the number of domain attributes in a relation (Figure 5). An optimal join order for Flat Storage might be suboptimal for Domain Storage if some of the relations have a significant number of domain attributes.

Consider the healthcare database schema [6] and some sample queries in Figure 6.

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 Figures 7 and 8 respectively. Alongwith every operator is given the size of its result. We observe that (Query Execution Plan) QEP 3

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 6. Sample schema and queries

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 becomes a must for such complicated queries. However, [6] uses a single nested loop join plan for all the queries.

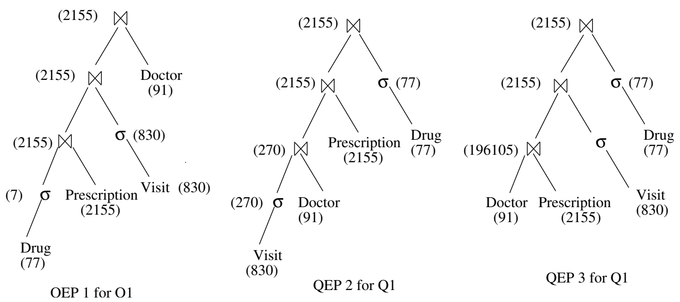

Figure 7. Query execution plans for Q1

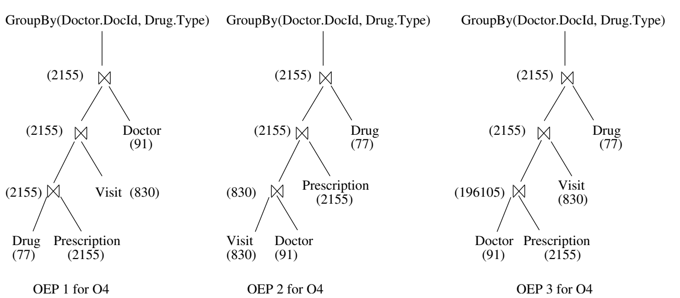

Figure 8. Query execution plans for Q4

3.2. Our Cost based Resource Cognizant Query Optimizer

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 operator order is evaluated using the System-R dynamic programming approach [18]. When we have left-deep tree and a fully pipelinable schedule, pushing down selections becomes 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. Non-leaf selections are evaluated in the join algorithms together with the join filters.

3.2.1 Operator evaluation schemes

The schemes for evaluating an operator use different amount of memory and have different cost. All the schemes result from the fact that left-deep tree is used as query plan and can be implemented using the well known *Iterator Model* [11]. 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[9]. The schemes for join operator are nested loop, index nested loop, hash join and join using *Join Index*. For aggregation, we have nested loop aggregation and buffered aggregation where we store the aggregate values. More information about the operator schemes is available at [19].

3.2.2 Benefit/Size ratio 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 as shown in Figure 9b. 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.

$$min(o) = s_{min}$$
$$\forall i, 1 \leq i \leq n : Cost(s_i) \leq Cost(s_{min}), \quad Memory(s_i) \geq Memory(s_{min})$$

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

$$Benefit(s_i) = Cost(s_{min}) - Cost(s_i)$$
$$Size(s_i) = Memory(s_i) - Memory(s_{min})$$

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. Figure 9a shows the benefit and size points for an operator. In general, the benefit of a

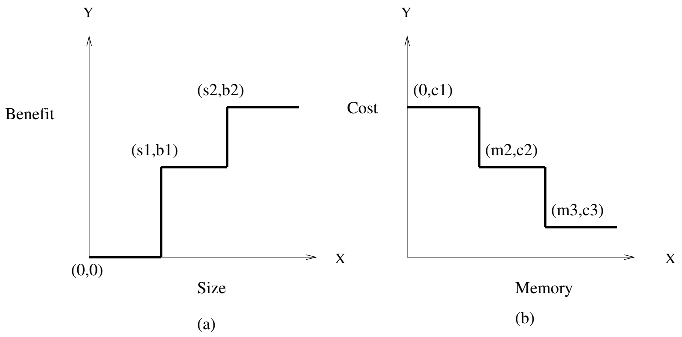

Figure 9. (a) Benefit,Size points for an operator (b) Operator cost function

scheme increases with the amount 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.

3.2.3 Modified 2-phase memory allocation

In the 2-phase approach, while determining the optimal plan in phase 1, maximum memory is allocated to all the operators. With this (maximum) amount of memory in hand, for every operator in the plan, the best scheme is determined in phase 1. 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 $feasible$[1]. 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 $feasible$ ones. So, schemes chosen in phase 1 and schemes chosen after phase 2 need not be the same. Clearly, optimal division of memory to the operators should involve selecting the best $feasible$ 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 operator order by choosing the least cost scheme for every operator. However, in phase 2, it chooses the best $feasible$ scheme for every operator depending on the memory allocated to it.

[1]A scheme is $feasible$ if its memory requirements can be met with the available memory

3.2.4 Exact Memory Allocation

A 2-phase exact solution to the memory allocation problem is proposed in [12]. 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 (i.e., the amount of memory to be given to each of the two operators). 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 optimization done in [12] is traditional 2-phase optimization. Hence, we need to modify the algorithms proposed in [12] to be able to use them in our system. [12] provides a procedure *OptMerge* (see Appendix) that constructs a superoperator cost function from piecewise linear operator cost functions. *OptMerge* is used to divide memory optimally among piece wise linear cost functions. [12] also proves that optimal division of memory for piece wise linear functions takes place only at change-over points.

The exact solution can thus be used in our context with the following modification. 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. Our 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 procedure *OptMerge*. 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 since 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 (i.e. *OptMerge*) is $O(z^2 \log z)$ where $z = max(n_i, n_j)$ [12]. 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 the cost value, memory value, and the division of memory at that point. Assuming an integer to store them and I as integer size,

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

$MemAllocate(M_{total})$
{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 10. Pseudo Code: MemAllocate

$GetBestScheme(M_{avail})$
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 11. Pseudo Code: 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 12. Pseudo Code: RecomputeBenefits

3.2.5 Heuristic Memory Allocation

The time and space complexity of the exact algorithm is very high and some of the handheld devices might not be able to cope with these. So, we need a heuristic solution to the memory allocation problem, one with a reduced time and space complexity.

We propose the following heuristic solution. Given the available memory, the heuristic determines 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 feasible scheme. We repeat this process until all the operators have been allocated memory. Our heuristic is as follows:

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

Our heuristic solution is inspired by the greedy algorithm for cache management in [17]. The procedure *MemAllocate* that allocates the memory is shown in Figure 10. 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 11) and *RecomputeBenefits* (Figure 12). 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 query. 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}. The required memory of the best scheme, i.e. $Memory(s_{best})$ is allocated to the operator o_{best} and M_{avail} is recomputed accordingly. s_{best} and all the schemes of o_{best} which need less memory than $Memory(s_{best})$ are removed from the list of schemes. The benefit and size values of the schemes of o_{best} which need more memory (and have more benefit) than $Memory(s_{best})$ need to be recomputed. 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}, determines which operator scheme has the highest benefit/size ratio among all the schemes of all operators that are feasible 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 feasible with M_{avail}.

Recomputation of Benefits

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}.

s_{best} need not be the final best 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 13 shows an example of benefit and size recomputation. Scheme 1 has the highest $Benefit/Size$ ratio but Scheme 2 has more benefit than Scheme 1. Initially, Scheme 1 will be selected and values for Scheme 2 will be recomputed. Now later during allocation, if $(s2 - s1)$ amount of memory is available and Scheme 2 has the highest $Benefit/Size$, Scheme 2 will be selected.

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.

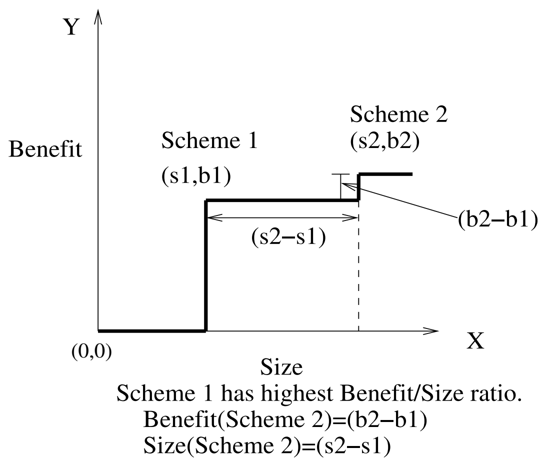

Figure 13. Benefit and Size Recomputation

4 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. [6] measure the performance of their query engine for smartcards on a Pentium-486 computer varying the system parameters. [7] measure the performance on a calibrated platform. However, the database will be eventually 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 can expect to get real performance numbers.

Model	Storage Cost
Flat Storage	31K
ID Storage	28.5K
ID Join Index	18K

Figure 14. Storage Costs

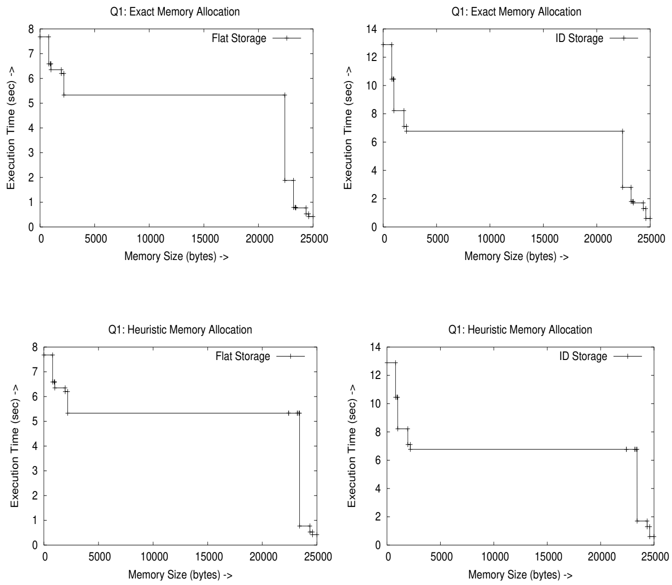

Figure 15. Performance Results for Query Q1

4.1. Experimental setup

We implemented our database system on the Simputer [4], a 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 like music player running on it.

For performance testing, we need schema, data, and queries which suit the needs of a handheld device. We select the healthcare schema and queries used in [6] and described in Figure 6 since healthcare applications are common in handheld devices. The dataset taken was a relatively large one (Doctor(91), Drug(77), Prescription(2155) , Visit(830)) leading to a 31K database. The datasets used

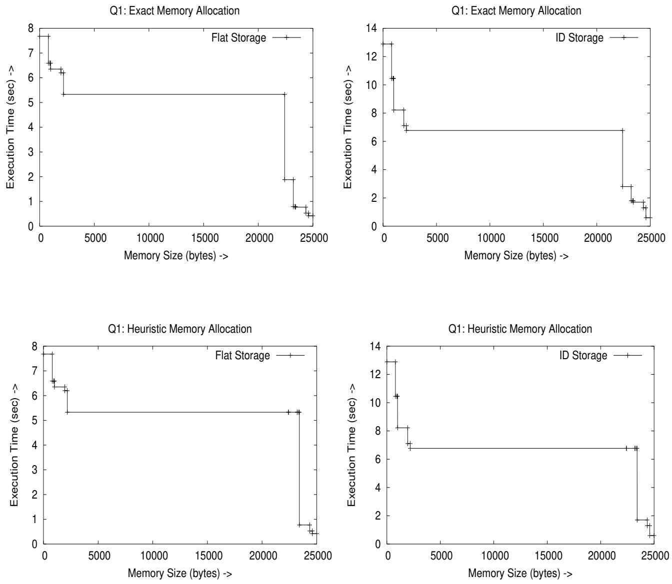

Figure 16. Performance Results for Query Q4

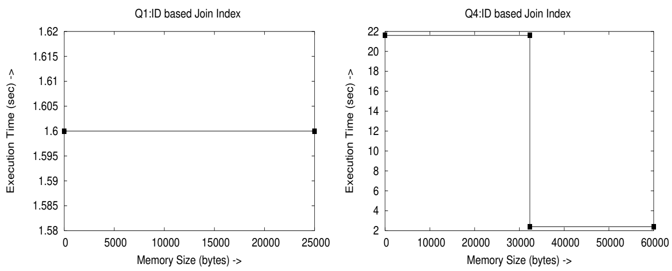

Figure 17. Performance Results using ID Join Index

are the same as [6]. We stored the data in three different storage schemes, Flat Storage, ID Storage, and ID based Join Index. The domain attributes are Doctor.Name, Drug.Name, and Drug.Type. The cardinalities of Doctor and Drug are less compared to Visit and Prescription. Hence, we expect ID Storage to reduce storage cost but the saving will not be very high. With ID based Join Index, we expect the storage cost to reduce considerably since relations Prescription and Visit have foreign keys. Figure 14 shows the storage costs for all the models.

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 executed all the 4 queries on the Simputer using the dataset. We measure the absolute response time of each query, varying the amount of memory available for execution. We optimize both the queries using the exact memory allocation method and the heuristic memory allocation method. Figure 15 and Figure 16 show the results for query Q1 and query Q4 with Flat and ID Storage.Figure 17 shows the results with Join Index. Since there is an extra cost for projection in ID Storage we expect the numbers for ID Storage to be higher than Flat Storage. The joins in both the queries are primary key-foreign key joins and hence we expect ID based Join Index scheme to reduce the query execution time considerably.

4.2. Observations

The storage cost saving with ID Storage was 2.5K since the cardinalities of the relations having domain attributes was not very high. However, with ID based Join Index the saving was around 13K which is a significant saving for a small device. The storage cost numbers justify our guidelines mentioned in Figure 4.

The response time was the highest with the query plan for zero memory and least with maximum memory. In all the graphs, the query plans change at a finite number of points. These points are the memory division points. The query plans 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 [6] for two important reasons (i) As mentioned earlier, [6] 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 7.6s to 0.4s for Flat Storage and 12.8s to 0.6s 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 7. 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. 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 $Doctor$ has a domain on $Name$.

For the aggregate query Q4 the response times ranged from 44.4s to 2.1s for Flat Storage and 73.5s to 2.76s 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 8. The groupby operator at the root was the most costly operator and influenced the execution time the most. As was the case in Q1, the heuristic algorithm generated the same query plan as the exact algorithm barring a few points. With increasing memory, query plans became less costly. 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.

The influence of ID based Join Index on the query response times was significant. Only the groupby operator needed memory as joins were done by traversing the Join Index. Q1 was executed in 1.6s. The times for query Q4 were 21.6s with zero memory and 2.4s with memory buffered aggregation. Thus, the response time reduced significantly for both the queries. The heuristic and exact allocation algorithm generated the same query plan in this case.

The performance results clearly suggest that query optimization is a must for complex queries. Our techniques chose the best possible plan given the available memory and the underlying storage model. 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. When ID Storage is used, the response time increase by a small factor. Hence, the best way to store the domain attributes is ID Storage. ID based Join Index is the best way to represent a primary key-foreign key relationship since not only it saves storage cost but also speeds up query processing.

Integration with an existing application

We integrated the database system with an existing application. AQUA is an online database backed discussion forum developed at Media Lab Asia, IIT Bombay [1]. The application is also meant to be used in mobile hosts. We ported AQUA on the Simputer and used our database to store the data and query it.

5. Conclusion

Lightweight computing devices are increasingly flooding many aspects of our life. As new applications appear, the need for embedded database support arise in various forms of such devices. The main constraints are limited stable storage and main memory. Different handheld devices come with different amount of computing resources, the database techniques should use the limited resources most efficiently. Storage models that save storage space need to be designed. Queries involve complex joins and aggregations thus leading to the need of query optimization. Optimal memory allocation among the database operators is needed to guarantee the best query execution plan.

In this paper, we proposed ID Storage, a new storage model that leads to a considerable saving in storage space. We showed how cost based query optimization becomes essential for complicated queries. We present an exact and heuristic algorithm to allocate memory among the database operators. Our performance analysis show that heuristic memory allocation works almost as fine as the exact allocation algorithm except for a few points. 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. The query engine guarantees the best query execution plan for all types of handheld devices.

As part of future work, 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.

References

[1] AQUA, Media Lab Asia IIT Bombay. http://www.mlasia.iitb.ac.in/aaqua.htm.

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

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

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

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

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

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

[8] A. Datta, D. VanderMeer, K. Ramamritham, and B. Moon. Applying Parallel Processing Techniques in Data Warehousing and OLAP. In *VLDB*, 1999.

[9] D. J. DeWitt, R. H. Katz, F. Olken, L. D. Shapiro, M. R. Stonebraker, and D. Wood. Implementation Techniques for Main Memory Database Systems. In *ACM SIGMOD*, 1984.

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

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

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

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

[14] 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.

[15] 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.

[16] J. Rao and K. A. Ross. Cache Conscious Indexing for Decision-Support in Main Memory. In *VLDB*, 1999.

[17] P. Roy. *Multi-Query Optimization and Applications*. PhD thesis, Indian Institute of Technology - Bombay, 2001.

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

[19] R. Sen. Second stage report. www.cse.iitb.ac.in/rajkumar/report.pdf.

[20] P. Seshadri. Honey I Shrunk the DBMS. In *ACM SIGMOD*, 1999.

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