Cracking the Database Store

Martin Kersten Stefan Manegold

CWI, Kruislaan 413, 1098 SJ Amsterdam, The Netherlands

{Martin.Kersten,Stefan.Manegold}@cwi.nl

Abstract

Query performance strongly depends on finding an execution plan that touches as few superfluous tuples as possible. The access structures deployed for this purpose, however, are non-discriminative. They assume every subset of the domain being indexed is equally important, and their structures cause a high maintenance overhead during updates. This approach often fails in decision support or scientific environments where index selection represents a weak compromise amongst many plausible plans.

An alternative route, explored here, is to continuously adapt the database organization by making reorganization an integral part of the query evaluation process. Every query is first analyzed for its contribution to break the database into multiple pieces, such that both the required subset is easily retrieved and subsequent queries may benefit from the new partitioning structure.

To study the potentials for this approach, we developed a small representative multi-query benchmark and ran experiments against several open-source DBMSs. The results obtained are indicative for a significant reduction in system complexity with clear performance benefits.

1 Introduction

The ultimate dream for a query processor is to touch only those tuples in the database that matter for the production of the query answer. This ideal cannot be achieved easily, because it requires upfront knowledge of the user's query intent.

In OLTP applications, all imaginable database subsets are considered of equal importance for query processing. The queries mostly retrieve just a few tuples without statistically relevant intra-dependencies. This permits a physical database design centered around index accelerators for individual tables and join-indices to speed up exploration of semantic meaningful links.

In decision support applications and scientific databases, however, it is a priori less evident what subsets are relevant for answering the -mostly statistical- queries. Queries tend to be ad-hoc and temporarily localized against a small portion of the databases. Data warehouse techniques, such as star- and snowflake schemas and bit-indices, are the primary tools to improve performance [Raf03].

In both domains, the ideal solution is approximated by a careful choice of auxiliary information to improve navigation to the database subset of interest. This choice is commonly made upfront by the database administrator and its properties are maintained during every database update. Alternatively, an automatic index selection tool may help in this process through analysis of the (anticipated) work load on the system [ZLLL01, ACK+04]. Between successive database reorganizations, a query is optimized against this static navigational access structure.

Since the choice of access structures is a balance between storage and maintenance overhead, every query will inevitably touch many tuples of no interest. Although the access structures often permit a partial predicate evaluation, it is only after the complete predicate evaluation that we know which access was in vain.

In this paper we explore a different route based on the hypothesis that access maintenance should be a byproduct of query processing, not of updates. A query is interpreted as both a request for a particular database subset and as an advice to *crack* the database store into smaller *pieces* augmented with an index to access them. If it is unavoidable to touch Una-interesting tuples during query evaluation, can we use that to prepare for a better future?

To illustrate, consider a simple query `select * from R where R.a <10` and a storage scheme that requires a full table scan, i.e. touching all tuples to select those of interest. The result produced in most systems is a stream of qualifying tuples. However, it can also be interpreted as a task to fragment the table into two pieces, i.e. apply horizontal fragmentation. This operation does not come for free, because the new table incarnation should be written back to persistent store and its properties stored in the catalog. For example, the original table can be replaced by a UNION TA-

Proceedings of the 2005 CIDR Conference

BLE [MyS] or partitioned table over all its pieces in Oracle, DB2, and Microsoft SQL-Server. The subsequent query optimizer step now has to deal with a fragmented table.

The key question is why this seemingly simple approach has not been embraced? Possible answers are: the overhead of continuous cracking a table is utterly expensive, the catalog of pieces and their role in query plan generation leads to an explosion in the search space, and there is no reference benchmark to study its effect in a multi-query scenario under laboratory conditions. Although database developers' wisdom may indeed point into the right direction for existing database systems, our research into next generation kernels calls for exploring new territories.

This paper's contributions are threefold: (i) it introduces a database organization scheme based on *cracking*, (ii) it introduces a multi-query benchmark to analyze the cracking scheme, and (iii) evaluates a prototype implementation of the key algorithms.

The experiments are run against out-of-the-box versions of a few open-source databases. The results provide a glimpse of the conditions to make cracking a success and the bottlenecks encountered in current DBMS offerings. The prospect of embedding cracking in a database kernel is further studied in the context of MonetDB[Mon]. Although this system takes an off-beat approach to physical organize the database and its processing, it illustrates the potentials in a software area we control.

The remainder of this paper is organised as follows. In Section 2, we scout the notion of database crackers in the context of the lowest denominator of database access, i.e. table scans. The cracker algorithm, administration, and optimization issues are reviewed in Section 3. Section 4 introduces a characterisation of multi-query streams for performance evaluations. An initial performance outlook using open-source database systems is given in Section 5.

2 Cracking the Database Store

The departure taken to invest in database reorganization in the critical path of ordinary query processing is based on a small experiment described in Section 2.1. An outlook on the performance challenge in long query sequences is presented in Section 2.2

2.1 Table Scans

Table scans form the lowest access level of most database kernels. They involve a sequential read of all tuples followed by predicate evaluation. The qualified tuples are moved to a result table, passed onward to the next operator in a query evaluation pipeline, or sent to a GUI. The expected gross performance is easy to predict. The source operand has to be read once, and the fragment of interest has to be written back to persistent store (or shipped to the application front-end).

For a query with selectivity factor σ and table size of N tuples, we know that $(1-\sigma)\ N$ tuples do not contribute to the final result. Unfortunately, to know which tuples disqualify, the predicate has to be evaluated against each tuple first. This raises the question at how much additional cost could we retain this information by concurrently fragmenting the table into pieces and building a catalog of table fragments. An experiment against existing systems provides an outlook of interest for the remainder.

Consider a table R[int,int] with 1 M tuples against which we fire a sequence of range queries

```
INSERT INTO newR
SELECT * FROM R    WHERE R.A>=low AND R.A<high
```

with varying selectivity. Figure 1 illustrates the response time encountered for (a) materialization into a temporary table, (b) sending the output to the front-end, and (c) just counting the qualifying tuples.[1] Aside from very small tables, the performance of materialization is linear in the size of the fragment selected; a key observation used in most cost-models for query optimization. For large tables it becomes linear in the number of disk IOs.

The performance figures also highlight the relative cost of the basic operations. Storing the result of a query in a new system table (a) is expensive, as the DBMS has to ensure transaction behavior. Sending it to the front-end (b) is already faster, although the systems behave quite differently on this aspect. Finally, the cost of finding qualifying tuples itself (c) is cheap in some systems.

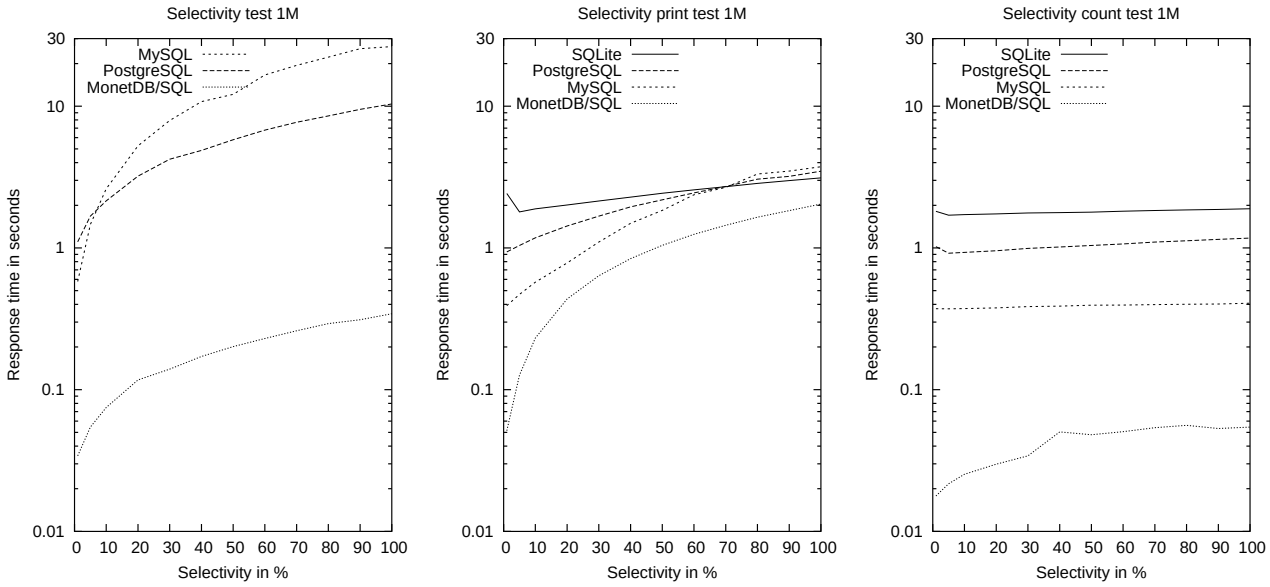
These differences are crucial in pursuing the cracking approach. It already indicates that the marginal overhead for writing a reorganized table back into the database store might be acceptable for relatively low selectivity factors when the result has to be delivered to the front-end anyway. Conversely, if the query is only interested in a count of qualifying tuples, it does not make sense to store the fragment at all. The system catalog maintenance would be too high.

Furthermore, when seen from the perspective of a single query with low selectivity factor, one is not easily tempted to repartition the table completely, a performance drop of an order of magnitude may be the result. A simple experiment suffices to reconsider this attitude.

2.2 An Outlook

Assume that we are confronted with an application that continuously fires range queries. Hundreds of them are issued in rapid succession and each query splits the fragments touched as part of the process. The catalog system — supported by an in-memory datastructure — keeps track of them as a partitioned table. With time progressing the retrieval speed would increase dramatically and the per-query

[1] All experiments are run on a dual Athlon 1400 processor with 1GB memory system. The systems considered are MonetDB, MySQL with ISAM backend, PostgreSQL, and SQLite. They ran out-of-the-box without any further optimization. Experiments against commercial systems show similar behavior.



(a) (b) (c)

Figure 1. Response time for relational algebra operations

overhead dwindles to a small fraction. In essence, this process is an incremental buildup of a search accelerator, driven by actual queries rather than database updates and guidelines given by a DBA. With proper engineering the total CPU cost for such an incremental scheme is in the same order of magnitude as sorting, plus the catalog maintenance cost.

The actual performance impact of this continual database reorganization strongly depends on the database volatility and query sequence. For a full table scan, we need N reads and σN writes for the query answer. Furthermore, in a cracker approach we may have to write all tuples to their new location, causing another $(1-\sigma)N$ writes; an investment which is possibly hard to turn to our advantage.

A small-scale simulation provides the following outlook. Consider a database represented as a vector where the elements denote the granule of interest, i.e. tuples or disk pages. From this vector we draw at random a range with fixed σ and update the cracker index. During each step we only touch the pieces that should be cracked to solve the query.

Figure 2 illustrates the fractional overhead in terms of writes for various selectivity factors using a uniform distribution and a query sequence of up to 20 steps. Selecting a few tuples (1%) in the first step generates a sizable overhead, because the database is effectively completely rewritten. However, already after a query sequence of 5 steps and a selectivity of 5%, the writing overhead due to cracking has dwindled to less than the answer size.

Figure 3 illustrates the corresponding accumulated overhead in terms of both reads and writes. The baseline (=1.0) is to read the vector. Above the baseline we have lost performance, below the baseline cracking has become beneficial. Observe that the break-even point is already reached after a handful of queries.

An alternative strategy (and optimal in read-only settings) would be to completely sort or index the table upfront, which would require $Nlog(N)$ writes. This investment would be recovered after $log(N)$ queries. Beware, however, that this only works in the limited case where the query sequence filters against the same attribute set. This restriction does not apply to cracking, where each and every query initiates breaking the database further into (irregular) pieces.

The experiment raises a number of questions. What kind of application scenarios would benefit from the cracking approach? How can the processing overhead of cracking in a real DBMS be reduced? What are the decisive factors in deciding on the investments to be made? What are the effects of updates on the scheme proposed? And, finally, how does the performance compare to the more traditional approach of secondary index maintenance? In the sequel, we can only address parts of these questions, leaving many desirable research questions for the future.

3 A Database Cracker Architecture

In this section, we introduce the architecture of a database cracker component. It is positioned between the semantic analyzer and the query optimizer of a modern DBMS infrastructure. As such, it could be integrated easily into existing systems, or used as a pre-processing stage before query processing. Section 3.1 introduces a class of database crackers, followed by the cracker index in Section

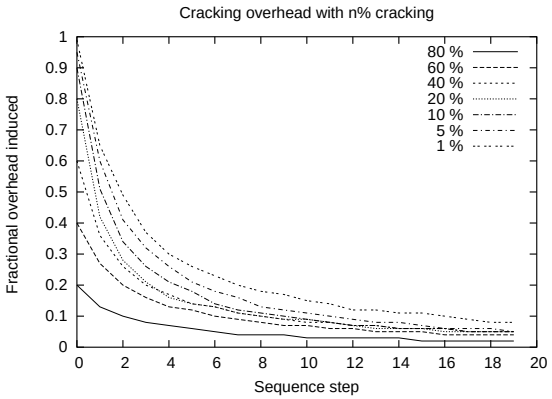


Figure 2. Cracking overhead

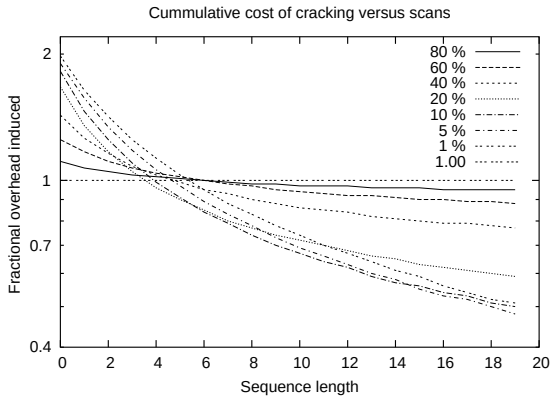


Figure 3. Accumulated overhead

3.2. In Section 3.4, we sketch the algorithms needed in a DBMS and, Section 3.3 identifies the effect on the query optimizer and plan generation.

3.1 Cracker Concepts

Informally, database crackers are queries that break a relational table into multiple, disjoint pieces. They are derived during the first step of query optimization, i.e. the translation of an SQL statement into a relational algebra expression. Without loss of generality, we assume that a query is in disjunctive normal form and each term organized as the expression:

$$\pi_{a_0,\ldots,a_k}\gamma_{grp}\sigma_{pred}(R_1 \bowtie R_2 \ldots R_{m-1} \bowtie R_m) \qquad (1)$$

The selection predicates considered are simple (range) conditions of the form $attr \in [low, high]$ or $attr\ \theta\ cst$ with θ in $\{<, <=, >, >=, =, !=\}$. For simplicity, the (natural-) join sequence is a join-path through the database schema. The γ operator denotes an aggregate grouping (GROUP BY).

This representation by no means implies a query evaluation order. It forms the basis to localize and extract the database crackers only. A traditional query optimizer is called upon in the second phase of the query evaluation process to derive an optimal plan of action.

The selection predicates and projection list form the first handle for cracking the database. It has been known for a long time, that they can be used to construct a horizontally/vertical fragmented table for distributed processing [OV91]. Once chosen properly, it will significantly improve performance both using parallel processing and being able to early filter tuples of no-interest to a query. It is this latter aspect we exploit.

Relational algebra operations, like $\pi_{attr}(R)$, $\sigma_{pred}(R)$, or $R \bowtie S$, suggest natural and intuitive ways to split their input table(s) into two fragments each, on containing the "interesting" tuples and the other one the "non-interesting" tuples. A projection $\pi_{attr}(R)$, for instance, suggests to vertically split a table such that one piece contains all attributes from the projection list *attr*, and the other one contains all attributes of *R* that are not in *attr*. Likewise, a (natural) join $R \bowtie S$ suggest to horizontally split each table in two pieces, one containing all tuples that find matches in the other relation, and a second that contains all tuples that do not find a match in the join. Obviously, the piece of either table is simply made-up by the semi-join with the other table.

As discussed earlier, a selection $\sigma_{pred}(R)$ suggests to horizontally split *R* into two pieces, where the first consists of all tuples that fulfill the predicate, and the non-qualifying tuples are gathered in the second piece. For single-side range predicates on only one attribute ($attr\ \theta\ cst$), the resulting pieces do have the "nice" property, that the values of the selection attribute form a consecutive range within each piece. However, this property gets lost in case of point-selections ($attr\ \theta\ cst$ with θ in $\{=, !=\}$) and double-sided range predicates $attr \in [low, high]$. To re-gain the consecutive ranges property, we propose a second version a selection-cracking that yields three pieces: $attr < low$, $attr \in [low, high]$, and $attr > high$ In this scenario, point-selections can be viewed as double-sided range selections with $low == high$. Beware, that this property globally holds only for the first cracking step on a "virgin" table. Once a table has been cracked, subsequent selection cracking maintain the consecutiveness only locally within the previous pieces.

Finally, group-by operation merely produce an n-way partitioning based on singleton values.

This leads to the following cracker definitions for a relational model and illustrated graphically for Ψ*large Diamond*(R) in Figure 4:

- Ψ-*cracking* The cracking operation $\Psi(\pi_{attr}(R))$ over an n-ary relation *R* produces two pieces
  $P_1 = \pi_{attr}(R)$,
  $P_2 = \pi_{attr(R)-attr}(R)$.
- Ξ-*cracking* The cracking operation $\Xi(\sigma_{pred}(R))$ over an n-ary relation *R* produces two pieces in case of $pred \equiv attr\ \theta\ cst, \theta \in \{<, <=, >, >=\}$ $P_1 = \sigma_{pred}(R)$, $P_2 = \sigma_{\neg pred}(R)$;

and three pieces in case of $pred \equiv attr \in / \notin [low, high]$
$P_1 = \sigma_{attr<low}(R)$,
$P_2 = \sigma_{attr\in[low,high]}(R)$,
$P_3 = \sigma_{high<attr}(R)$.

- $\diamondsuit$-*cracking* The cracking operation $\diamondsuit(R \bowtie S)$ over two relations produces four pieces,
$P_1 = R \ltimes S$,
$P_2 = R \backslash (R \ltimes S)$,
$P_3 = S \ltimes R$,
$P_4 = S \backslash (S \ltimes R)$.

- Ω-*cracking* The cracking operation $\Omega(\gamma_{grp}R)$ produces a collection $\{P_{\vec{i}}\}_{\vec{i} \in \pi_{grp}R} = \sigma_{grp=\vec{i}}(R)$.

All four crackers are *loss-less*, i.e., the original table can be reconstructed from the pieces generated by each cracker. For Ξ, $\diamondsuit$, and Ω the *inverse* of cracking is simply a union of all pieces. For Ψ, we assume that each vertical fragment includes (or is assigned) a unique (i.e., duplicate-free) surrogate (`oid`), that allows simple reconstruction by means of a natural 1:1-join between the surrogates of both pieces.

3.2 Cracker Index

Cracking the database into pieces should be complemented with information to reconstruct its original state and result tables, which means we have to administer the lineage of each piece, i.e. its source and the Ξ, Ψ, $\diamondsuit$ or Ω operators applied.

This information can be stored in the system catalog, as it involves a partitioned table, or as a separate access structure. The former approach does not seem the most efficient track, given the way a partitioned table is administered in current DBMS implementations. Each creation or removal of a partition is a change to the table's schema and catalog entries. It requires locking a critical resource and may force recompilation of cached queries and update plans.

Instead, we propose a cracker index, which for each piece keeps track of the (min,max) bounds of the (range) attributes, its size, and its location in the database. The location is a new table or a view over the table being cracked. The boundary information is maintained for all ordered attributes. It is used to navigate the cracker index and it provides key information for the query cost model.
Consider the query sequence:

```
select * from R where R.a<10;
select * from R,S where R.k=S.k and R.a<5;
select * from S where S.b>25;
```

Figure 5 illustrates the cracker index produced. Relation R is broken into two pieces R[1] with R.a>= 10 and R[2] with R.a<10 tuples. In the next step term R.a<5 limits its search to just R[2] and only R[4] is used to initiate the $\diamondsuit$ cracker. R[6] and S[3] contain the elements that will join on attribute k. The last query now has to inspect both S[3] and S[4] because nothing has been derived about attribute b. Since we are interested in all attributes, cracking causes only two pieces in each step. Observe that R can be reconstruction by taking the union over R[1], R[3], R[5], and R[6], and S using S[5], s[6], s[7], and S[8].

The cracker index can be represented as a global graph datastructure or organized on a per table basis. Its size can be controlled by selectively trimming the graph applying the inverse operation to the nodes.

Observe that for cracking we do not assume a priori knowledge of the query sequence. Each time a query arrives, it is validated against the cracker index and may initiate a change. For all but simple queries this calls for a difficult decision, because a relational algebra expression provides many cracking options. An alternative cracker index is shown in Figure 6 where the Ξ and $\diamondsuit$ operations in the second query are interchanged.

This phenomenon calls for a cracking optimizer which controls the number of pieces to produce. It is as yet unclear, if this optimizer should work towards the smallest pieces or try to retain large chunks. A plausible strategy is to optimize towards many pieces in the beginning and shift to the larger chunks when we already have a large cracker index.

Whatever the choice, the cracker index grows quickly and becomes the target of a resource management challenge. At some point, cracking is completely overshadowed by cracker index maintenance overhead. A foreseeable track is to introduce a separate process to coalesce small pieces into larger chunks, but which heuristic works best with minimal amount of work remains an open issue.

3.3 Optimizer Issues

The cracker strategy leads to an explosion in the number of table fragments. For example, a Ξ cracker over an ordered domain breaks a piece into three new pieces. As pieces become smaller, the chance of being broken up also reduces. Furthermore, for each query only the pieces at the predicate boundaries should be considered for further cracking. Likewise, the $\diamondsuit$-cracker produces two pieces for each operand involved in a join. This means for a linear k-way join $4(k-1)$ pieces are added to the cracker index. The Ω cracker adds another $2^{|g|}$ pieces for a grouping over g attributes. Overall these numbers suggest a possible disastrous effect on an optimizer. Especially if it builds a complete set of alternative evaluation plans to pick the best strategy upfront. We hypothesize that an optimizer in the cracker context has an easier job. This can be seen as follows.

The prime task of an optimizer is to align the operators in such a way that the minimal number of intermediate results are produced. The cracking index helps here, because each query sub-plan will be highly focused on a portion of the target result and involve mostly small tables. They hardly

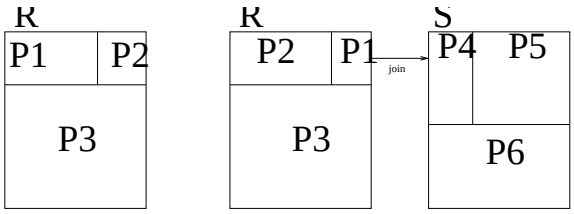


Figure 4. Ξ and ◇ cracking

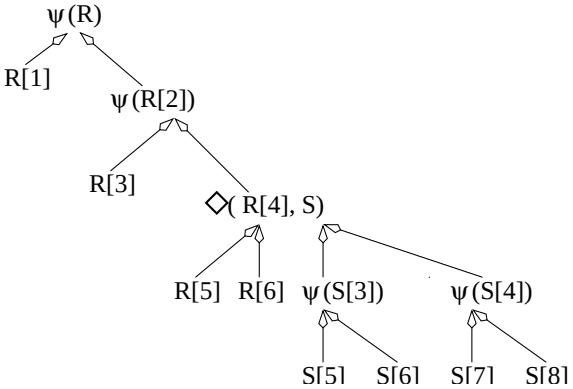


Figure 5. Cracker lineage

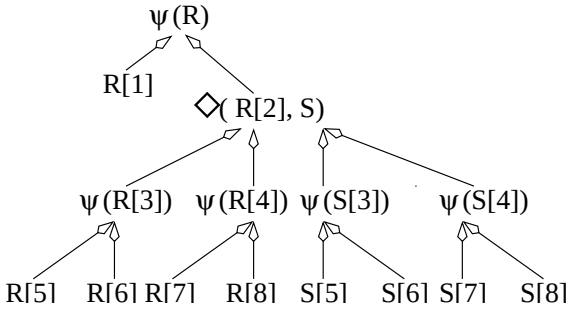


Figure 6. Alternate cracker lineage

touch unwanted tuples and, therefore, need only be materialized when the user retrieves their content.

The Ξ cracker effectively realizes the select-push-down rewrite rule of the optimizer. The pieces of interest for query evaluation are all available with precise statistics. The ◇ cracker effectively builds a semi-join-index, splitting each input in two pieces containing those tuples that find a match in the join, and those that do not, respectively. The first piece can by used to calculate the join without caring about non-matching tuples, the second piece contains the additional tuples for an outer-join. The Ω cracker clusters the elements into disjoint groups, such that subsequent aggregation and filtering are simplified. Since we do not assume any additional index structure, the optimizer merely has to find an efficient scheme to combine the pieces. Localization cost has dropped to zero, due to the cracker, and the optimizer can focus on minimization of intermediate results only.

3.4 Cracking Algorithms

Ideally, the database is cracked with minimal CPU cost, and with minimal additional storage overhead. Database extensibility and piggybacking on the query evaluation are the prime tools considered for this.

3.4.1 Crackers in a Query Processor

Cracking can be used in a conventional DBMS where it takes the form of continual data reorganization, e.g. partitioning decisions at each query step. Rather than relying upon the help of a separate database partitioning tool, one could piggyback cracking over normal query evaluations as follows.

Most systems use a Volcano-like query evaluation scheme [Gra93]. Tuples are read from source relations and passed up the tree through filter-, join-, and projection-nodes. The cracker approach can be readily included in this infrastructure.

The Ξ-cracker can be put in front of a filter node to write unwanted tuples into a separated piece. The tuples reaching the top of the operator tree are stored in their own piece. Taken together, the pieces can be used to replace the original tables. The storage overhead, however, is the total size of the base tables used during query processing. The overhead for tuple inserts can be reduced using the knowledge that no integrity checking is needed. The transaction processing overhead due to moving tuples around can, however, not be ignored. A similar technique can be applied to the Ψ, ◇ and Ω crackers.

3.4.2 Crackers in an Extensible DBMS

A cracking scheme can also be implemented in an extensible DBMS as a new accelerator structure. We provide details for one such system.

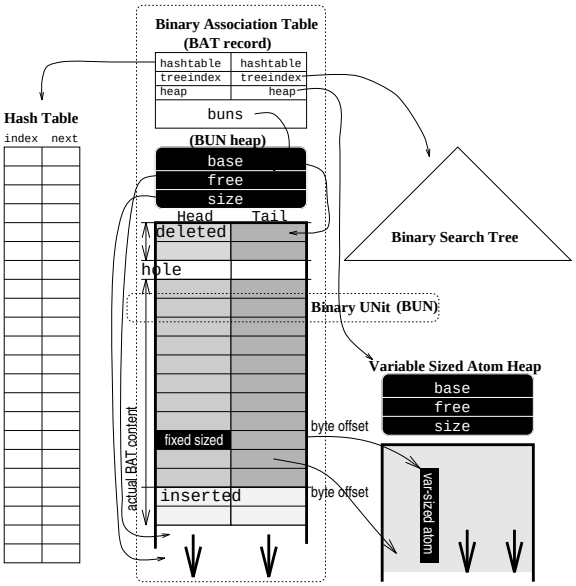


Figure 7. BAT layout

The MonetDB[2], developed at CWI, is our major platform for experimentation in core database technologies. It is built around a storage structure for binary relations only, called Binary Association Tables (BATs) [BMK00]. The physical structure of a BAT is shown in Figure 7. It is a contiguous area of fixed-length records with automatically maintained search accelerators. Variable length elements are collected in separate storage areas, called the heaps. New elements are appended and elements to be deleted are moved to the front until transaction commit. The BATs are memory mapped from disk and the memory management unit of the system is used to guarantee transaction isolation.

N-ary relational tables are mapped by MonetDB's SQL compiler into a series binary tables with attributes `head` and `tail` of type `bat`[*oid*,*tpe*], where *oid* is the surrogate key and *tpe* the type of the corresponding attribute. For more details see: [BMK00, MBK02].

The cracking algorithms are collected into a user defined extension module, which can be moved transparently between the SQL compiler and the existing kernel by overloading the key algebraic operators: `select`, `join`, and `aggregate`.

The Ξ cracker algorithm takes a value-range and performs a shuffle-exchange sort over all tuples to cluster them according to their tail value. The shuffling takes place in the original storage area, relying on the transaction manager to not overwrite the original until commit.

With the data physically stored in a single container, we can also use MonetDB's cheap mechanism to slice portions from it using a BAT view. A BAT view appears to the user as an independent binary table, but its physical location is determined by a range of tuples in another BAT. Consequently, the overhead incurred by catalog management is less severe. The MonetDB BATviews provide a cheap representation of the newly created table. Their location within the BAT storage area and their statistical properties are copied to the cracker index. Of course, only pieces that need to be cracked are considered.

The $\diamond$ cracker algorithm is a slight modification of the existing join algorithms. Instead of producing a separate table with the tuples being join-compatible, we shuffle the tuples around such that both operands have a consecutive area with matching tuples. The Ω operation can be implemented as a variation of the Ξ cracker.

The research challenge on the table is to find a balance between cracking the database into pieces, the overhead it incurs in terms of cracker index management, query optimization, and query evaluation plan. Possible cut-off points to consider are the disk-blocks, being the slowest granularity in the system, or to limit the number of pieces administered. If the cracker dictionary overflows, pieces can be merged to form larger units again, but potentially defeating the benefit of ignoring unwanted tuples altogether.

Finding answers to these questions call for a laboratory setting to study the contribution of the design parameters. Therefore, we have formulated a multi-query benchmark generation kit, introduced below. It can be used in a much wider setting to study progress in multi-query optimization.

4 Application areas

The application areas foreseen for database cracking are data warehouses and scientific databases. Datawarehouses provide the basis for datamining, which is characterized by lengthly query sequences zooming into a portion of statistical interest [BRK98].

In the scientific domain, the databases may be composed of a limited number of tables with hundreds of columns and multi-million rows of floating point numbers. For example, the tables keep track of timed physical events detected by many sensors in the field [SBN+99]. In addition, the database contains many derived tables, e.g. to represent model fitting experiments. It is topped with a version scheme to keep track of the lineage of the tables being managed. In practice, a coarse-grain fragmentation strategy is used to break up a terabyte database into pieces of a few tens of gigabytes each.

For studying database crackers, we step away from application specifics and use a generic, re-usable framework. The space of multi-query sequences is organized around a few dimensions based on idealistic user behavior. Within this setting, we distinguish between a *homerun*, a *hiking*, and a *strolling* user profile. They are introduced in more detail below.

Homeruns

The *homerun* user profile illustrates a user zooming into a specific subset of σN tuples, using a multi-step query re-

[2]MonetDB, an opensource DBMS http://www.monetdb.com

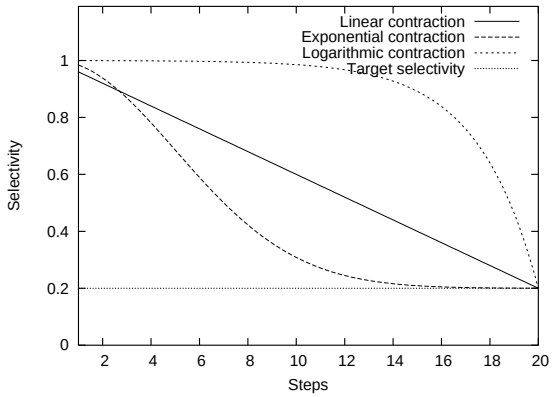


Figure 8. Selectivity distribution ($\sigma = 0.2, k = 20$)

finement process. It represents a hypothetical user, who is able to consistently improve his query with each step taken, such that he reaches his final destination in precisely k steps (the length of the query sequence). Zooming presupposes lack of a priori knowledge about the database content, which leads to initially ill-phrased queries.

There are many ways to model this convergence process. In our benchmark, we consider three extreme cases: linear, logarithmic, and exponential convergence. Under a linear convergence model, a user is consistently able to remove a constant number of tuples from an intermediate answer to reach his goal. The selectivity factor at each step of the sequence can be described by a selectivity distribution function $\rho(i,k,\sigma)$, which in this case produces a subset of size $(1-i(1-\sigma)/k)N$ tuples at the i-th step in the query sequence.

A more realistic scenario is a sequence where, in the initial phase, the candidate set is quickly trimmed and where, in the tail of the sequence, the hard work takes place by fine-tuning the query expression to precisely fit the target set. This scenario can be modeled with an exponential distribution function: $\rho(i,k,\sigma) = \sigma + (1-\sigma)e^{-(1-\sigma)/2ki^2}$.

The complementary case is a logarithmic distribution function, where the quick reduction to the desired target takes place in the tail of the sequence. This is modeled by the function $\rho(i,k,\sigma) = 1-(1-\sigma)e^{-(1-\sigma)/2(k-i)}$. The behavior of these models is illustrated in Figure 8.

The *homerun* models a sequence of range refinements and a monotonously reducing answer sets. It models a case, where the user already has knowledge of the whereabouts of the target set, but needs to fine-tune the parameters for its retrieval.

Hiking

In many sessions, though, discovering attributes of interest is an integral part of the job. This means that a user will explore not only different range selections on a fixed attribute set, but will try out different attributes or navigate through join relationships inspired by the database schema. It also occurs in situations where the database is continuously filled with stream/sensor information and the application has to keep track or localize interesting elements in a limited window.

In the *hiking* profile, we assume that such shifts in focus are not random. Instead, the answer sets of two consecutive queries partly overlap. They steer the search process to the final goal. We assume that our ideal user is able to identify at each step precisely σN tuples, which aligns with an interaction sequence driven by sampling and top-n queries. The overlap between answer sets reaches 100% at the end of the sequence. The selectivity distribution functions can be used to define overlap by $\delta(i,k,\sigma) = \rho(i,k,0)$.

Strolling

The base line for a multi-query sequence is when the user has no clue where to look for specifically. He samples the database in various directions using more or less random steps until he stumbles upon a portion that can be explored further with a *homerun* or *hiking* strategy.

In this situation, we only assume ultimately retrieving σN tuples from the database. There is no zooming behavior and there is no exploration of the navigational semantic structure.

The selectivity function shown in Figure 8 can be used to generate meaningful sequences. A convergence sequence can be generated using the i-th selectivity factor to select a random portion of the database. Alternatively, we can use the function as a selectivity distribution function. At each step we draw a random step number to find a selectivity factor. Picking may be with or without replacement. In all cases, the query bounds of the value range are determined at random.

Multi-Query Sequences

The databases used for experimentation are generated by the DBtapestry program. The output of this program is an SQL script to build a table with N rows and α columns. The value in each column is a permutation of the numbers 1..N. SQL updates can be used to mold the tapestry table to create one with the data distributions required for detailed experimentation.

The tapestry tables are constructed from a small seed table with a permutation of a small integer range, which is replicated to arrive at the required table size, and, finally, shuffled to obtain a random distribution of tuples.

The dimensions for multi-query sequences against the tapestry table can be combined in many ways, giving a large space to pick from. This space can be concisely defined as follows:

DEFINITION The query sequence space can be characterised by the tuple

$$MQS(\alpha,N,k,\sigma,\rho,\delta) \qquad (2)$$

where α denotes the table arity
N the cardinality of the table
k the length of the sequence to reach the target set
σ the selectivity factor of the target set
ρ the selectivity distribution function $\rho(i,k,\sigma)$
δ the pair-wise overlap as a selectivity factor over N

A study along the different dimensions provides insight in the ability of a DBMS to cope with and exploit the nature of such sequences. For the remainder of this paper, we use the search space to assess our cracking approach.

5 Experimentation

A cracking approach is of interest if and only if its embedding within mature relational systems is feasible performance wise. Although the necessary data structures and algorithms can readily be plugged into most architectures, it is unclear whether the envisioned benefits are not jeopardized by fixed overhead of fragment management or other global database management tasks. To obtain an outlook on the global benefits, we conducted a series of experiments against PostgreSQL[Pos], MySQL[MyS], and MonetDB[Mon]. The former two are open-source traditional n-ary relational engines, while MonetDB's design is based on the binary relational model.

For the preliminary experiments, we used a tapestry table of various sizes, but with only two columns. It is sufficient to highlight the overheads incurred and provides a baseline for performance assessment.

In Section 5.1, we introduce an approach for SQL based systems. For extensible databases it may be possible to introduce a new module. This is illustrated using MonetDB in Section 5.2.

5.1 Crackers in an SQL Environment

To peek into the future with little cost, we analyze the crackers using an independent component at the SQL level using the database engine as a black box. To illustrate, consider a database with the relational table:

```
create table R( k integer, a integer);
```

A Ξ cracker *attr* θ *constant* breaks it into two pieces. As SQL does not allow us to move tuples to multiple result tables in one query, we have to resort to two scans over the database.

```
select into frag001
   r.k, r.a from r where pred(r.a);
select into frag002
   r.k, r.a from r where not pred(r.a);
```

The cost components to consider are: i) creation of the cracker index in the system catalog, ii) the scans over the relation and (iii) writing each tuple to its own fragment. This involves rudimentary database operations, whose performance is already summarized in Figure 1. They illustrate that materialization of a temporary table in a DBMS involves a sizable overhead. It ranges from 40 to 1200 ms per table. On top of this, the DBMS needs anywhere between 3 to 250 ms/10000 tuples to evaluate the predicate and create the tuple copy.

For example, consider a query with a selectivity of 5% ran against MySQL and delivering the information to the GUI. Such a query would cost in this database setup around 0.5 second. Storing the same information in a temporary table adds another 1.5 seconds. This is not all, because cracking requires the original table to be broken into two pieces, effectively raising the total response time to around 10 seconds. The investment of 9.5 seconds during this cracking step is hard to turn into a profit using less scan cost in the remainder of the sequence. To put it in perspective, sorting the table on this attribute alone took about 250 seconds.

The simulation of the Ψ, $\diamond$ and Ω operators in an SQL setting require multiple scans as well.

During result construction, the pieces localized in the cracker index should be combined to produce the result table. This requires fast table unions and join operations to undo Ξ and $\diamond$ cracking. Since cracking may produce many tables with just a few attributes and a few tuples linked with surrogate keys, we have to rely on the DBMS capabilities to handle large union expressions and long foreign-key join sequences efficiently to construct the result tables. Although join algorithms and join optimization schemes belong to the most deeply studied areas, an experiment against our database clearly showed a bottleneck when cracking is deployed in a traditional context.

For example, consider the relational table above, comprised of only two columns and just a million elements. The tuples form random integer pairs, which means we can 'unroll' the reachability relation using lengthly join sequences. We tested the systems with sequences of up to 128 joins. The results are shown in Figure 9. It demonstrates that the join-optimizer currently deployed (too) quickly reaches its limitations and falls back to a default solution. The effect is an expensive nested-loop join or even breaking the system by running out of optimizer resource space.

The practical consequence is that cracking in older systems is confined to tables breaking it up in just a small collection of vertical fragments. Otherwise the join implementation becomes a stand in the way to glue the partial results together. A notable exception is MonetDB, which is built around the notion of binary tables and is capable handling such lengthly join sequences efficiently.

Taken into account these performance figures and the baseline cost of primitive operators, it does not seem prudent to implement a cracker scheme within the current offerings. Unless one is willing to change the inner-most algorithms to cut down the overhead.

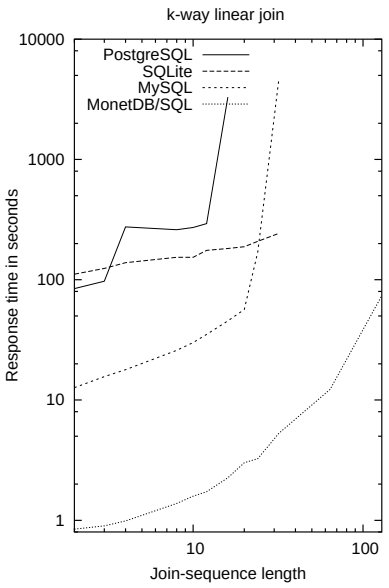


Figure 9. Linear join experiment

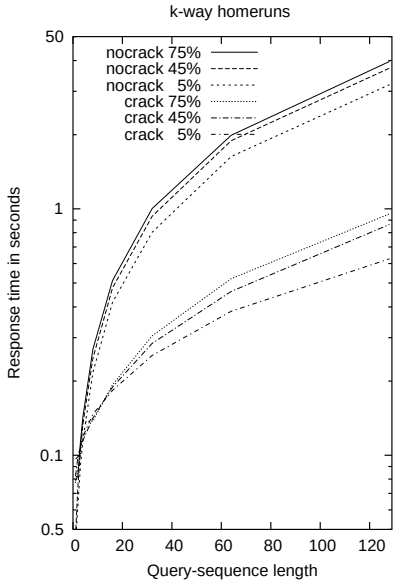


Figure 10. Homerun experiment (MonetDB)

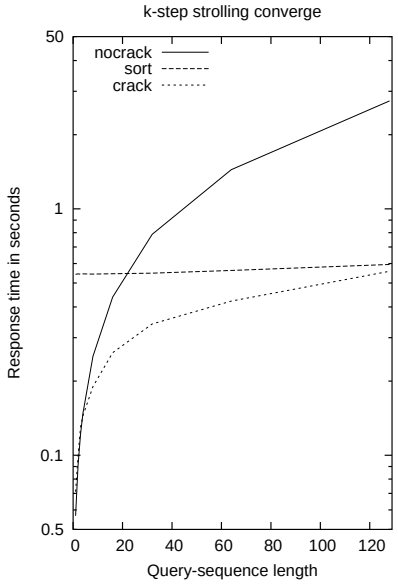


Figure 11. Random converge experiment (MonetDB)

5.2 Crackers in MonetDB

A cracker module has been implemented to assess whether the cost structure encountered at the SQL level can be countered. The module implements the Ξ and $\diamondsuit$ crackers and relies on the systems efficient memory management scheme to guarantee transaction safety during cracking. The cracker index is organized as a decorated interval tree. Each table comes with its own cracker index and they are not saved between sessions. They are pure auxiliary datastructures to speedup processing of queries over selected binary tables.

The initial experiments reported here were targeted at converging query sequences using both *homerun* and *strolling* strategies.

Homeruns

A cracking strategy is expected to work best when a multi-query sequence zooms into a target set and where answers to previous queries help to speedup processing. Figure 10 is an illustrative result we obtained using the multi-query benchmark. It illustrates the total response time for a linear homerun sequence of k ($<$128) steps. The selectivity factor indicates the target size to reach after those steps. The time for both with and without cracking support is shown. The latter merely results in multiple scans over the database and any performance gain is an effect of a hot table segment lying around in the DBMS cache.

The lines for cracking indicate its adaptive behavior. After a few steps it outperforms the traditional scans and ultimately leads to a total reduction time of a factor 4. Under a cracking strategy the response times for the individual steps also quickly reduce. It provides a response time of a nearly completely indexed table.

Strolling

The baseline test is to simulate a random walk through the database, i.e. the user (or multiple users) fire selection queries against the database without evident intra-query dependencies. Figure 11 is an illustrative example for the case where we use the selectivity distribution function to converge to a desired target size of 5%. Sequences up to 128 steps are executed and compared against the non-cracking approach, and the situation where in the first step we sort the table for fast lookup.

The results confirm the performance improvement over non-cracking. It also shows that cracking is a viable alternative to sorting (or secondary index construction) if the number of queries interested in the attribute is rather low. Investment in an index becomes profitable performance cost-wise when the query sequence exceeds 100 steps and randomly browses the database.

6 Related Research

The cracker approach is inspired by techniques from distributed database research [OV91], partial indexing [SS95], and clustered indices.

From distributed databases, we take derived-horizontal and vertical fragmentation to crack the database into smaller pieces. The cracking scheme, however, is not a one-time action. It is performed continuously and produces irregular

fragments upon need. The database partitioning scheme is a byproduct of each query. Their lineage is made available for dynamic adjustments and continual reorganizations.

Modern research in distributed databases, e.g. sensor networks and P2P systems, tend to extrapolate the traditional DDBMS design routes where a profile of interest is translated into advice on caching and locality of information. The balanced profiles of multiple users are defined by the DBA using a powerful description language[CGFZ03]. In a cracking scheme, the equivalent to profiles are the queries being fired and resource balancing takes place continuously.

Even in non-distributed systems, partitioning large tables has become an important technique to reduce index maintenance and to improved query responsiveness. For example, MySQL, DB2 and Oracle advice the DBA to specify range-partitioned tables for large datawarehouses. The cracker scheme extends this practice using a cracker index defined implicitly by query use, rather then by intervention of a DBA.

A second source of inspiration came from partial indices, an area largely neglected in database research. Already in 1989, Stonebraker identified the opportunity to incrementally built/maintain an index as part of the query execution phase [Sto89]. The technique has been implemented in Postgres, but appears hardly ever used. The prospects of partial indexing has been studied by Sheshadri and Swami using a simulation tool in [SS95]. It demonstrates convincingly the potential for improved performance and lower maintenance cost under a wide variety of parameter settings. However, the upfront knowledge required are database statistics and a workload. The cracker index can be considered a partial index; it is built as part of accessing portions of interest. It follows the idea of partial indexing by shifting the cost of maintenance to the query user.

Clustering tuples has long been an important technique to achieve high performance. Tuples within a single relation are grouped together using a cluster index, but also tuples from different tables may be clustered in the same disk page to speedup foreign-key joins. An interesting approach in this direction is presented in [Gra03], where the B-tree structure is extended to support dynamic reorganization. It might provide the proper setting to experiment with the crackers in a commercial system. The implementation of the MonetDB cracker module dynamically clusters tuples within the same table space. The experiments so far are confined to range-based clustering and the cracker index binds the fragments together into an interval tree structure. Clustering significantly reduces the IO cost and is known to be effective in a main-memory/cache setting as well [MBK02].

A solution to index selection, clustering and partition is addressed by emerging toolkits, such as DB2 Design Advisor [ZLLL01], Microsoft Database Tuning Advisor [ACK+04], and Oracle's SQL Tuning Advisor[Ora03]. They work from the premises that the future access pattern can be predicted from the past or from a mock-up workload. For most application settings this seems the right way to go. The cracking approach is advocated for the decision support and scientific database field, where interest in portions of the database is ad hoc, localized in time, and mostly unpredictable. The database size, both in terms of attributes and tuples, precludes creation of many indices. Moreover, the clients typically flock around a portion of the database for a limited period, e.g. the readings from multiple scientific devices for a star in our galaxy.

7 Summary and future research

To achieve progress in database systems research calls for challenging established implementation routes, e.g. by studying new design parameter constellations. The challenge put on the table here is: *Let the query users pay for maintaining the access structures*. Its realization is the notion of database *cracking*, where a query is first interpreted as a request to break the database into pieces organized by a cracker index. After cracking the database the query is evaluated using ordinary (distributed) query optimization techniques.

The cracker model leads to a simple, adaptive accelerator structure with limited storage overhead. The portion of the database that matters in a multi-query sequence is coarsely indexed. Only by moving outside this hot-set, investments are needed.

The cracker approach extends techniques developed in the context of distributed databases. However, these techniques have not (yet) been used — as far as we are aware — to its extreme as proposed here for managing the database in a central setting.

The basic experiments on traditional relational systems show that the cracking overhead is not neglectable. To be successful, the technique should be incorporated at the level where index structures are being maintained. Moreover, a main-memory catalog structure seems needed to reduce the large overhead otherwise experienced in maintaining a system catalog.

As the database becomes cracked into many pieces, the query optimizer is in a better position to discard portions of no interest for evaluation. A laboratory benchmark has been defined to pursue detailed performance studies. Focused multi-query sequences, typically generated by datamining applications [BRK98], in particular benefit from cracking.

Using database cracking as the leading force to improve performance in query dominant environment calls for many more in depth studies. The experiments reported indicate opportunities for novel query optimization strategies, complementary to materialized views currently being the predominant route to support multi-query sequences. The diminishing return on investment as the cracker index becomes too large calls for heuristics or learning algorithms to fuse pieces together. Finally, database cracking may proof a sound basis to realize self-organizing databases in a P2P environment.

Acknowledgments

This work was supported by the BRICKS project, a National program to advance long term research in computer science. The authors are thankful for comments on early drafts and contributions of the MonetDB development team.

References

[ACK+04] Sanjay Agrawal, Surajit Chaudhuri, Lubor Kollár, Arunprasad P. Marathe, Vivek R. arasayya, and Manoj Syamala. Database tuning advisor for microsoft sql server. In *Proc. of the Int'l. Conf. on Very Large Data Bases*, Toronto, Canada, August 2004. To appear.

[BMK00] P. Boncz, S. Manegold, and M. Kersten. Optimizing Database Architecture for the New Bottleneck: Memory Access (Extended Paper Published For Best-Of-VLDB'99). *The VLDB Journal*, 9(3):231–246, December 2000.

[BRK98] P. Boncz, T. Rühl, and F. Kwakkel. The Drill Down Benchmark. In *Proc. of the Int'l. Conf. on Very Large Data Bases*, pages 628–632, New York, NY, USA, June 1998.

[CGFZ03] Mitch Cherniack, Eduardo F. Galvez, Michael J. Franklin, and Stan Zdonik. Profile-driven cache management. In *International Conference on Data Engineering (ICDE)*, 2003.

[Gra93] G. Graefe. Query Evaluation Techniques for Large Databases. *ACM Computing Surveys*, 25(2):73–170, June 1993.

[Gra03] G. Graefe. Sorting and indexing with partitioned b-trees. In *Proceedings of the CIDR 2003 Conference*, Jan, 2003.

[MBK02] S. Manegold, P. Boncz, and M. Kersten. Optimizing Main-Memory Join On Modern Hardware. *IEEE Trans. on Knowledge and Data Eng.*, 14(4):709–730, 2002.

[Mon] MonetDB. http://www.monetdb.com.

[MyS] MySQL. http://www.mysql.com.

[Ora03] Oracle Corp. *Oracle8 SQL Reference*, 2003. http:// otn. oracle. com/ documentation/ database10g.html.

[OV91] A. Ozsu and P. Valduriez. *Principles of Distributed Database Systems*. Prentice Hall, Englewood Cliffs, NJ, USA, 1991.

[Pos] PostgreSQL. http://www.postgresql.com.

[Raf03] Maurizio Rafanelli, editor. *Multidimensional Databases: Problems and Solutions*. Idea Group, 2003.

[SBN+99] Arie Shoshani, Luis M. Bernardo, Henrik Nordberg, Doron Rotem, and Alex Sim. Multidimensional indexing and query coordination for tertiary storage management. pages 214–225, 1999.

[SS95] Praveen Seshadri and Arun N. Swami. Generalized partial indexes. In Philip S. Yu and Arbee L. P. Chen, editors, *Proceedings of the Eleventh International Conference on Data Engineering, March 6-10, 1995, Taipei, Taiwan*, pages 420–427. IEEE Computer Society, 1995.

[Sto89] Michael Stonebraker. The case for partial indexes. *SIGMOD Record*, 18(4):4–11, 1989.

[ZLLL01] Daniel Zilio, Sam Lightstone, Kelly Lyons, and Guy Lohman. Self-managing technology in ibm db2 universal database. In *Proceedings of the tenth international conference on Information and knowledge management*, pages 541–543. ACM Press, 2001.