Centrum voor Wiskunde en Informatica

REPORTRAPPORT

INS

Information Systems

INformation Systems

Generic database cost models for hierarchical memory systems

S. Manegold, P.A. Boncz, M.L. Kersten

REPORT INS-R0203 MARCH 31, 2002

CWI is the National Research Institute for Mathematics and Computer Science. It is sponsored by the Netherlands Organization for Scientific Research (NWO).
CWI is a founding member of ERCIM, the European Research Consortium for Informatics and Mathematics.

CWI's research has a theme-oriented structure and is grouped into four clusters. Listed below are the names of the clusters and in parentheses their acronyms.

Probability, Networks and Algorithms (PNA)

Software Engineering (SEN)

Modelling, Analysis and Simulation (MAS)

Information Systems (INS)

P.O. Box 94079, 1090 GB Amsterdam (NL)
Kruislaan 413, 1098 SJ Amsterdam (NL)
Telephone +31 20 592 9333
Telefax +31 20 592 4199

ISSN 1386-3681

Generic Database Cost Models for Hierarchical Memory Systems

Stefan Manegold
S.Manegold@cwi.nl

Peter Boncz
P.Boncz@cwi.nl

Martin L. Kersten
M.L.Kersten@cwi.nl

CWI
P.O. Box 94079, 1090 GB Amsterdam, The Netherlands

ABSTRACT

Accurate prediction of operator execution time is a prerequisite for database query optimization. Although extensively studied for conventional disk-based DBMSs, cost modeling in main-memory DBMSs is still an open issue. Recent database research has demonstrated that memory access is more and more becoming a significant—if not the major—cost component of database operations. If used properly, fast but small cache memories—usually organized in cascading hierarchy between CPU and main memory—can help to reduce memory access costs. However, they make the cost estimation problem more complex.

In this article, we propose a generic technique to create accurate cost functions for database operations. We identify a few basic memory access patterns and provide cost functions that estimate their access costs for each level of the memory hierarchy. The cost functions are parameterized to accommodate various hardware characteristics appropriately. Combining the basic patterns, we can describe the memory access patterns of database operations. The cost functions of database operations can automatically be derived by combining the basic patterns' cost functions accordingly.

To validate our approach, we performed experiments using our DBMS prototype Monet. The results presented here confirm the accuracy of our cost models for different operations.

Aside from being useful for query optimization, our models provide insight to tune algorithms not only in a main-memory DBMS, but also in a disk-based DBMS with a large main-memory buffer cache.

1998 ACM Computing Classification System: [H.2.4] Main-Memory Database Systems, Query Processing
Keywords and Phrases: main-memory databases, query optimization, cost modeling, memory access costs
Note: Work carried out under project INS1.2 “Database Architecture”.

1. Introduction

Database cost models provide the foundation for query optimizers to derive an efficient execution plan. Such models consist of two parts: a logical and a physical component. The former is geared towards estimation of the data volumes involved. Usually, statistics about the data stored in the database are used to predict the amount of data that each operator has to process. The underlying assumption is that a query plan that has to process less data will also consume less resources and/or take less time to be evaluated. The logical cost component depends only on the data stored in the database, the operators in the query, and the order in which these operators are to be evaluated (as specified by the query execution plan). Hence, the logical cost component is independent of the algorithm and/or implementation used for each operator.

The problem of (intermediate) result size estimation has been intensively studied in literature [PIHS96, IP99, RR98]. In this article, we focus on the physical cost component. Therefore, we assume a perfect oracle to predict the data volumes.

Given the data volumes, the physical cost component is needed to discriminate the costs of the various algorithms and implementations of each operator. The query optimizer uses this information to choose the most suitable algorithm and/or implementation for each operator.

Given the fact that disk-access used to be the predominant cost factor, early physical cost functions just counted the number of I/O operations to be executed by each algorithm [Gra94]. Any operation that loads a page from disk into the in-memory buffer pool or writes a page from the buffer back to disk is counted as an

I/O operation. However, disk systems depict significant differences in cost (in terms of time) per I/O operation depending on the access pattern. Sequentially reading or writing consecutive pages causes less cost per page than accessing scattered pages in a random order. Hence, more accurate cost models discriminate between random and sequential I/O. The cost for sequential I/O is calculated as the data volume[1] divided by the I/O bandwidth. The cost for random I/O additionally considers the seek latency per operation.

With memory chips dropping in price while growing in capacity, main memory sizes grow as well. Hence, more and more query processing work is done in main memory, trying to minimize disk access as far as possible in order to avoid the I/O bottleneck. Consequently, the contribution of pure CPU time to the overall query evaluation time becomes more important. Cost models are extended to model CPU costs, usually in terms of CPU cycles (scored by the CPU's clock speed to obtain the elapsed time).

CPU cost used to cover memory access costs [LN96, WK90]. This implicitly assumes that main memory access costs are uniform, i.e., independent of the memory address being accessed and the order in which different data items are accessed. However, recent database research has demonstrated that this assumption does not hold (anymore) [ADHW99, BMK99]. With hierarchical memory systems being used, access latency varies significantly, depending on whether the requested data can be found in (any) cache, or has to be fetch from main memory. The state (or contents) of the cache(s) in turn depends on the applications' access patterns, i.e., the order in which the required data items are accessed. Furthermore, memory latency has hardly improved over the last decade, while CPU speed is continuously experiencing an exponential growth. Hence, memory access has become a significant cost factor — not only for main memory databases — which cost models need to reflect.

In query execution, the memory access issue has been addressed by designing new cache-conscious data structures [RR99, RR00, ADHS01] and algorithms [SKN94, MBK00b]. On the modeling side, however, nothing has been published yet considering memory access appropriately.

In this article, we address the problem of how to model memory access costs of database operators appropriately. As it turns out to be quite complicated to derive proper memory access cost functions for various operations, we developed a new technique to automatically derive such cost functions. The basic idea is to describe the data access behavior of an algorithm in terms of a combination of basic access patterns (such as "sequential" or "random"). The actual cost function is then obtained by combining the patterns' cost functions (as derived in this article) appropriately. Using a unified hardware model that covers the cost-related characteristics of both main memory and disk access, it is straight forward to extend our approach to consider I/O cost as well. Gathering I/O and memory cost models into a single common framework is a new approach that simplifies the task of generating accurate cost functions.

In Section 2, we will discuss hierarchical memory systems and introduce our unified hardware model. Section 3 will present a simplified abstract representation of data structures and identify a number of basic access patterns to be performed on such data structures. Equipped with these tools, we will show how to specify the data access patterns of database algorithms by combining basic patterns. In Section 4 we will derive the cost function for our basic access patterns and Section 5 provides rules how to obtain the cost function of database algorithms from their representation introduced in Section 3. Section 6 contains some experimental results validating the obtained cost functions and Section 7 will draw some conclusions.

2. Hierarchical Memory Systems

2.1 Cache Memories

The fundamental principle of all cache architectures is "*reference locality*". The assumption is that at any time the CPU, respectively the program, repeatedly accesses only a limited amount of data that fits in the cache. Only the first access is "slow", as the data has to be loaded from main memory. We call this a *cache miss*. Subsequent accesses (to the same data or memory addresses) are then "fast" as the data is then available in the cache. We call this a *cache hit*. The fraction of memory accesses that can be fulfilled from the cache is called *cache hit rate*; analogously, the fraction of memory accesses that cannot be fulfilled from the cache is called *cache miss rate*.

[1] i.e., number of sequential I/O operations multiplied by the page size

Cache memories are often organized in *multiple cascading levels* between the main memory and the CPU. We refer to the individual caches as *first level* (*L1*) cache, *second level* (*L2*) cache, and so on. In nowadays computers, there are typically two or three cache levels. Often, L1 and L2 are integrated on the CPU's die, while L3—if present—is located on the system board. Caches become faster, but smaller, the closer they are to the CPU.

Caches are characterized by three major parameters: Capacity (C), Line Size (B), and Associativity (A):

Capacity (C). A cache's capacity defines its total size in bytes. Typical cache sizes range from 16 KB to 8 MB.

Line Size (B). Caches are organized in *cache lines*, which represent the smallest unit of transfer between adjacent cache levels. Whenever a cache miss occurs, a complete cache line (i.e., multiple consecutive words) is loaded from the next cache level or from main memory, transferring all bits in the cache line in parallel over a wide bus. This exploits spatial locality, increasing the chances of cache hits for future references to data that is ”closed to” the reference that caused a cache miss. Typical cache line sizes range from 16 bytes to 128 bytes.

Dividing the cache capacity by the cache line size, we get the *number of available cache lines* in the cache: $\# = C/B$. Cache lines are often also called *cache blocks*. We will use both terms as synonyms throughout this document.

Associativity (A). To which cache line the memory is loaded, depends on the memory address and on the cache's *associativity*. An *A-way set associative* cache allows to load a line in A different positions. If $A > 1$, some *cache replacement* policy chooses one from the A candidates. *Least Recently Used* (*LRU*) is the most common replacement algorithm. In case $A = 1$, we call the cache *direct-mapped*. This organization causes the least (virtually no) overhead in determining the cache line candidate. However, it also offers the least flexibility and may cause a lot of *conflict misses* (see below). The other extreme case are *fully associative* caches. Here, each memory address can be loaded to any line in the cache ($A = \#$). This avoids conflict misses, and only *capacity misses* (see below) occur as the cache capacity gets exceeded. However, determining the cache line candidate in this strategy causes a relatively high overhead that increases with the cache size. Hence, it is feasible only for smaller caches. Current PCs and workstations typically implement 2-way to 8-way set associative caches.

Cache misses can be classified into the following disjoint types [HS89]:

Compulsory. The very first reference to a cache line always causes a cache miss, which is hence classified as a compulsory miss. The number of compulsory misses obviously depends only on the data volume and the cache line size.

Capacity. A reference that misses in a fully associative cache is classified as a capacity miss because the finite sized cache is unable to hold all the referenced data. Capacity misses can be minimized by increasing the temporal and spacial locality of references in the algorithm. Increasing cache size also reduces the capacity misses because it captures more locality.

Conflict. A reference that hits in a fully associative cache but misses in an A-way set associative cache is classified as a conflict miss. This is because even though the cache was large enough to hold all the recently accessed data, its associativity constraints force some of the required data out of the cache prematurely. For instance, alternately accessing just two memory addresses that “happen to be” mapped to the same cache line will cause a conflict cache miss with each access. Conflict misses are the hardest to remove because they occur due to address conflicts in the data structure layout and are specific to a cache size and associativity. Data structures would, in general, have to be remapped so as to minimize conflicting addresses. Increasing the associativity of a cache will decrease the conflict misses.

2.2 *Memory Access Costs*

We identify the following three aspects that determine memory access costs. For simplicity of presentation, we assume 2 cache levels in this section. Generalization to an arbitrary number of caches is straight forward.

Latency is the time span that passes after issuing a data access until the requested data is available in the CPU. In hierarchical memory systems, the latency increases with the distance from the CPU. Accessing data that is already available in the L1 cache causes *L1 access latency* (λ_{L1}), which is typically rather small (1 or 2 CPU cycles). In case the requested data in not found in L1, an *L1 miss* occurs, additionally delaying the data access by *L2 access latency* ($\lambda_{\rm L2}$) for accessing the L2 cache. Analogously, if the data is not yet available in L2, an *L2 miss* occurs, further delaying the access by *memory access latency* ($\lambda_{\rm Mem}$) to finally load the data from main memory. Hence, the total latency to access data that is in neither cache is $\lambda_{\rm Mem} + \lambda_{\rm L2} + \lambda_{\rm L1}$. As L1 accesses cannot be avoided, we assume in the remainder of this article, that L1 latency is included in the pure CPU costs, and regard only memory latency and L2 latency as explicit memory access costs. As mentioned above, all current hardware actually transfers multiple consecutive words, i.e., a complete cache line, during this time.

When a CPU requests data from a certain memory address, modern DRAM chips supply not only the requested data, but also the data from subsequent addresses. The data is then available without additional address request. This feature is called *Extended Data Output* (EDO). Anticipating sequential memory access, EDO reduces the effective latency. Hence, we actually need to distinguish two types of latency for memory access. *Sequential access latency* ($\lambda^{\rm s}$) occurs with sequential memory access, exploiting the EDO feature. With random memory access, EDO does not speed up memory access. Thus, *random access latency* $\lambda^{\rm r}$ is usually higher than sequential latency.

Bandwidth is a metric for the data volume (in megabytes) that can be transfered between CPU and main memory per second. Bandwidth usually decreases with the distance from the CPU, i.e., between L1 and L2 more data can be transfered per time than between L2 and main memory. We refer to the different bandwidths as *L2 access bandwidth* ($\beta_{\rm L2}$) and *memory access bandwidth* ($\beta_{\rm Mem}$), respectively. In conventional hardware, the memory bandwidth used to be simply the cache line size divided by the memory latency. Modern multiprocessor systems typically provide excess bandwidth capacity $\beta' \geq \beta$. To exploit this, caches need to be *non-blocking*, i.e., they need to allow more than one outstanding memory load at a time, and the CPU has to be able to issue subsequent load requests while waiting for the first one(s) to be resolved. Further, the access pattern needs to be sequential, in order to exploit the EDO feature as described above.

Indicating its dependency on sequential access, we refer to the excess bandwidth as *sequential access bandwidth* ($\beta^{\rm s} = \beta'$). We define the respective *sequential access latency* as $\lambda^{\rm s} = B/\beta^{\rm s}$. For *random access latency* as described above, we define the respective *random access bandwidth* as $\beta^{\rm r} = B/\lambda^{\rm r}$. For better readability, we will simply use plain λ and β (i.e., without $^{\rm s}$ respectively $^{\rm r}$) whenever we refer to both sequential and random access without explicitly distinguishing between them.

On some architectures, there is a difference between read and write bandwidth, but this difference tends to be small. Therefore, we do not distinguish between read and write bandwidth in this article.

Address translation. For data access, logical virtual memory addresses used by application code have to be translated to physical page addresses in the main memory of the computer. In modern CPUs, a *Translation Lookaside Buffer* (*TLB*) is used as a cache for physical page addresses, holding the translation for the most recently used pages (typically 64). If a logical address is found in the TLB, the translation has no additional costs. Otherwise, a *TLB miss* occurs. The more pages an application uses (which also depends on the often configurable size of the memory pages), the higher the probability of TLB misses.

The actual *TLB miss latency* ($l_{\rm TLB}$) depends on whether a system handles a TLB miss in hardware or in software. With software-handled TLB, TLB miss latency can be up to an order of magnitude larger than with hardware-handled TLB. Hardware-handled TLB fetches the translation from a fixed memory structure, which is just filled by the operating system. Software-handled TLB leaves the translation method entirely to the operating system, but requires trapping to a routine in the operating system kernel on each TLB miss. Depending on the implementation and hardware architecture, TLB misses can therefore be more costly even than a main-memory access. Moreover, as address translation often requires accessing some memory structure, this can in turn trigger additional memory cache misses.

We will treat TLBs just like memory caches, using the memory page size as their cache line size, and calculating their (virtual) capacity as $number\_of\_entries \times page\_size$. TLBs are usually fully associative. Like caches, TLBs can be organized in multiple cascading levels.

For TLBs, there is no difference between sequential and random access latency. Further, bandwidth is irrelevant for TLBs, because a TLB miss does not cause any data transfer.

description	unit	symbol
cache name (level)	-	Li
cache capacity	[bytes]	C_i
cache block size	[bytes]	B_i
number of cache lines	-	$\#_i = C_i/B_i$
cache associativity	-	A_i
sequential access		
access bandwidth	[bytes/ns]	β^{s}_{i+1}
access latency	[ns]	$\lambda^{\mathsf{s}}_{i+1} = B_i/\beta^{\mathsf{s}}_{i+1}$
miss latency	[ns]	$l^{\mathsf{s}}_i = \lambda^{\mathsf{s}}_{i+1}$
miss bandwidth	[bytes/ns]	$b^{\mathsf{s}}_i = \beta^{\mathsf{s}}_{i+1}$
random access		
access latency	[ns]	$\lambda^{\mathsf{r}}_{i+1}$
access bandwidth	[bytes/ns]	$\beta^{\mathsf{r}}_{i+1} = B_i/\lambda^{\mathsf{r}}_{i+1}$
miss bandwidth	[bytes/ns]	$b^{\mathsf{r}}_i = \beta^{\mathsf{r}}_{i+1}$
miss latency	[ns]	$l^{\mathsf{r}}_i = \lambda^{\mathsf{r}}_{i+1}$

Table 1: Characteristic Parameters per Cache Level ($i \in \{1, \ldots, N\}$)[2]

2.3 Unified Hardware Model

Summarizing our previous discussion, we describe a computers memory hardware as a cascading hierarchy of N levels of caches (including TLBs). We add an index $i \in \{1, \ldots, N\}$ to the parameters described above to refer to the respective value of a specific level. The relation between access latency and access bandwidth then becomes $\lambda_{i+1} = B_i/\beta_{i+1}$. To simplify the notation, we exploit the dualism that an access to level $i + 1$ is caused a miss on level i. Introducing the *miss latency* $l_i = \lambda_{i+1}$ and the respective *miss bandwidth* $b_i = \beta_{i+1}$, we get $l_i = B_i/b_i$. Each cache level is characterized by the parameters given in Table 1.[2] In [MBK00b], we presented a system independent C program to measure these parameters on any computer hardware.[3] We point out, that these parameters also cover the cost-relevant characteristics of disk accesses. Hence, viewing main memory (e.g., a database system's buffer pool) as cache for I/O operations, it is straight forward to include disk access in this hardware model. Where appropriate, we use level $N + 1$ as synonym for main memory respectively secondary storage.

3. THE IDEA

Our recent work on main-memory database algorithms suggests that memory access cost can be modeled by estimating the number of cache misses $\mathbf{M}$ and scoring them with their respective miss latency l [MBK02]. This approach is similar to the one used for detailed I/O cost models. The hardware discussion above shows, that also for main-memory access, we have to distinguish between sequential and random access patterns. However, in contrary to disk access, we now have multiple levels of cache with varying characteristics. Hence, the challenge is to predict the number and kind of cache misses *for all cache levels*. Our hypothesis is, that we can treat all cache level individually, though equally, and calculate the total cost as the sum of the cost for all levels:

$$T_{\mathrm{Mem}} = \sum_{i=1}^{N} (\mathbf{M}^{\mathsf{s}}_i \cdot l^{\mathsf{s}}_i + \mathbf{M}^{\mathsf{r}}_i \cdot l^{\mathsf{r}}_i). \tag{3.1}$$

With the hardware modeled as described in the previous section and the hardware parameters measured by our calibration tool [MBK00b], the remaining challenge is to estimate the number and kind of cache misses

[2] We assume, that costs for L1 cache accesses are included in the CPU costs, i.e., λ_1 and β_1 are not used and hence undefined.

[3] The program called *Calibrator* is freely available from download for our web site: `http://monetdb.cwi.nl`

per cache level for various database algorithms. The task is similar to estimating the number and kind of I/O operations in traditional cost models. However, our goal is to provide a generic technique for predicting cache miss rates of various database algorithms. Nevertheless, we want to sacrifice as little accuracy as possible to this generalization.

To achieve the generalization, we introduce two abstractions. Both of them are driven by the goal to keep the models as simple as possible, but as detailed as necessary. Hence, we try to ignore any details that are not significant for our purpose (predicting cache miss rates) and only focus on the relevant parameters. Our first abstraction is a unified description of data structures. We call it *data regions*. The second are *basic data access patterns*. The following paragraphs will present both abstractions in detail.

3.1 Data Regions

We model data structures as *data regions*. $\mathbb{D}$ denotes the set of data regions. A data region $R \in \mathbb{D}$ consists of $R.n$ *data items* of size $R.w$ (in bytes). We call $R.n$ the *length* of region R, $R.w$ its *width* and $||R|| = R.n \cdot R.w$ its *size*. Further, we define the *number of cache lines covered by* R as $|R|_B = \lceil ||R||/B \rceil$, and the *number of data items that fit in the cache* as $|C|_{R.w} = \lceil C/R.w \rceil$.

A (relational) database table is hence represented by a region R with $R.n$ being the table's cardinality and $R.w$ being the tuple size (or width). Similarly, more complex structures like trees are modeled by regions with $R.n$ representing the number of nodes and $R.w$ representing the size (width) of a single node.

3.2 Basic Access Patterns

Data access patterns vary in their referential locality and hence in their cache behavior. Thus, not only the cost (latency) of cache misses depend on the access pattern, but also the number of cache misses that occur. Each database algorithm describes a different data access pattern. This means, each algorithm requires an individual cost function to predict its cache misses. Deriving each cost function ”by hand” is not only exhaustive and time consuming, but also error-prone. Our hypothesis is that we only need to specify the cost functions of a few basic access patterns. Given these basic patterns and their cost functions, we could describe the access patterns of database operations as combinations of basic access patterns, and derive the resulting cost functions automatically.

In order to identify the relevant basic access patterns, we have to analyze the data access characteristics of database operators, first. We classify database operations according to the number of operands.

Unary operators—such as, e.g., table scan, selection, projection, sorting, hashing, aggregation, or duplicate elimination—read data from one input region and write data to one output region. Data access can hence be modeled by two cursors, one for the input and one for the output. The input cursor traverses the input region sequentially. For table scan, selection, and projection, the output cursor also simply progresses sequentially with each output item. When building a hash table, the output cursor ”hops back and forth” in a non-sequential way. In practice, the actual pattern is not completely random, but rather depends on the physical order and attribute value distribution of the input data as well as on the hash function. In our case, i.e., knowing only the algorithm, but not the actual data, it is not possible to make more accurate (and usable) assumptions about the pattern described by the output cursor. Hence, we assume that the output region is accessed in a completely random manner. This assumption should not be too bad, as a ”good” hash function typically destroys any sorting order and tends/tries to level out skew data distributions.

Sort algorithms typically perform a more complicated data access pattern. In Section 6.2, we will present quick-sort as an example to demonstrate how such patterns can be specified as combinations of basic patterns. Aggregation and duplicate elimination are usually implemented using sorting or hashing. Thus, they perform the respective patterns.

Though also a unary operation, data partitioning takes a separate role. Again, the input region is traversed sequentially. However, modeling the output cursor's access pattern as purely random is too simple. In fact, we can do better. Suppose, we want to partition the input region into m output regions. Then, we know that the access within each region is sequential. Hence, we model the output access as a *nested* pattern. Each region is a separate *local cursor*, performing a sequential pattern. A single *global cursor* hops back and forth between the regions. Similar to the hashing scenario described before, the order in which the different region-cursors

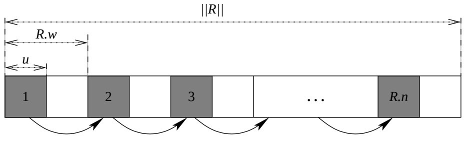


Figure 1: Single Sequential Traversal: $\mathsf{s\_trav}(R, u)$

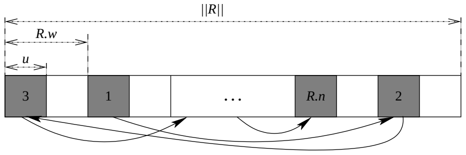


Figure 2: Single Random Traversal: $\mathsf{r\_trav}(R, u)$

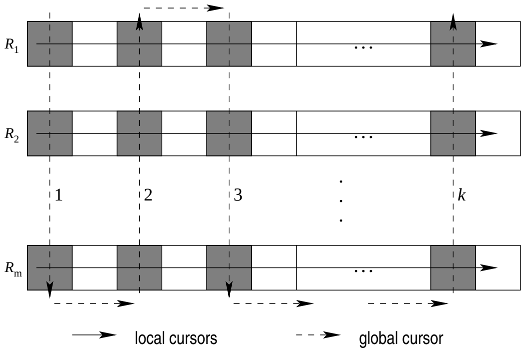


Figure 3: Interleaved Multi-Cursor Access: $\mathsf{nest}(R, m, \mathsf{s\_trav}(R, u), \mathtt{seq}, \mathtt{bi})$

are accessed—i.e., the global pattern—depends on the partitioning criterion (e.g., hash- or range-based) and the physical order and attribute value distribution of the input data. Again, it is not possible to model these dependencies in a general way without detailed knowledge about the actual data to process. Purely from the algorithm, we can only deduce a random order.

Concerning binary operations, we focus our discussion on join. The appropriate treatment of union, intersection and set-difference can be derived respectively. Binary operators have two inputs and a single output. In most cases, one input—we call it *left* or *outer* input—is traversed sequentially. Access to the other—*right* or *inner*—input depends on the algorithm and the data of the left input. A nested loop join performs a complete sequential traversal over the whole inner input for each outer data item. A merge join—assuming both inputs are already sorted—sequentially traverses the inner input once while the outer input is traversed. A hash join—provided there is already a hash table on the inner input—performs an "un-ordered" access pattern on the inner input's hash table. As discussed above, we assume a uniform random access.

From this discussion, we identify the following basic access patterns as eminent in the majority of relational algebra implementations.

single sequential traversal: $\mathsf{s\_trav}(R[, u])$

A sequential traversal sequentially sweeps over R, accessing each data item in R exactly once. The optional parameter u gives the number of bytes that are actually used of each data item. If not specified, we assume that all bytes are used, i.e., $u = R.w$. If specified, we require $0 < u \leq R.w$. u is used to model the fact that an operator, e.g., an aggregation or a projection (either as separate operator or in-lined with another operator), accesses only a subset of its input's attributes. For simplicity of presentation, we assume that we always access u consecutive bytes. Though not completely accurate, this is a reasonable abstraction in our case.[4] Figure 1 shows a sample sequential traversal.

repetitive sequential traversal: $\mathsf{rs\_trav}(r, d, R, [, u])$

A repetitive sequential traversal performs r sequential traversals over R after another. d specifies, whether all traversals sweep over R in the same direction, or whether subsequent traversals go in alternating directions. The first case—*uni-directional*—is specified by $d = \mathtt{uni}$. The second case—*bi-directional*—is specified by $d = \mathtt{bi}$.

single random traversal: $\mathsf{r\_trav}(R[, u])$

Like a sequential traversal, a random traversal accesses each data item in R exactly once, reading or writing u bytes. However, the data items are not accessed in the order they are stored, but rather randomly. Figure 2 depicts a sample random traversal.

[4] In case the u bytes are rather somehow spread across the whole item width $R.w$, say as k times u' bytes ($k \cdot u' = u$), one can replace $\mathsf{s\_trav}(R, u)$ by $\mathsf{s\_trav}(R', u')$ with $R'.w = R.w/k$ and $R'.n = R.n \cdot k$.

repetitive random traversal: $\mathsf{rr\_trav}(r, R[, u])$

A repetitive random traversal performs r random traversals over R after another. We assume that the permutation orders of two subsequent traversals are independent of each other. Hence, there is no point in discriminating uni-directional and bi-directional accesses, here. Therefore, we omit parameter *d*.

random access: $\mathsf{r\_acc}(r, R[, u])$

Random access hits r randomly chosen data items in R after another. We assume, that each data item may be hit more than once, and that the choices are independent of each other. Even with $r \geq R.n$ we do not require that each data item is accessed at least once.

interleaved multi-cursor access: $\mathsf{nest}(R, m, \mathcal{P}, O[, D])$

A nested multi-cursor access models a pattern where R is divided into m (equal-sized) sub-regions. Each sub-region has its own local cursor. All local cursors perform the same basic pattern, given by $\mathcal{P}$. O specifies, whether the global cursor picks the local cursors randomly ($O = \texttt{ran}$) or sequentially ($O = \texttt{seq}$). In the latter case, D specifies, whether all traversals of the global cursor across the local cursors use same direction ($D = \texttt{uni}$), or whether subsequent traversals use alternating directions ($D = \texttt{bi}$). Figure 3 shows a sample interleaved multi-cursor access.

3.3 Compound Access Patterns

Database operations access more than one data region, usually at least their input(s) and their output. This means, they perform more complex data access patterns than the basic ones we introduced in the previous section. In order to model these complex patterns, we now introduce *compound data access patterns*. Unless we need to explicitly distinguish between basic and compound data access patterns, we refer to both as data access patterns, or simply patterns. We use $\mathbb{P}_\mathrm{b}$, $\mathbb{P}_\mathrm{c}$, and $\mathbb{P} = \mathbb{P}_\mathrm{b} \cup \mathbb{P}_\mathrm{c}$ to denote the set of basic access pattern, compound access pattern, and all access pattern, respectively. We require $\mathbb{P}_\mathrm{b} \cap \mathbb{P}_\mathrm{c} = \emptyset$.

Be $\mathcal{P}_1, \ldots, \mathcal{P}_p \in \mathbb{P}$ ($p > 1$) data access patterns. There are two principle ways to combine two or more patterns. Either the patterns are executed *one after the other* or they are executed *concurrently*. We call the first combination *sequential execution* and denote it by operator $\oplus : \mathbb{P} \to \mathbb{P}$; the second combination represents *concurrent execution* and is denoted by operator $\odot : \mathbb{P} \to \mathbb{P}$. The result of either combination is again a (compound) data access pattern. Hence, we can apply $\oplus$ and $\odot$ repeatedly to describe complex patterns. By definition, $\odot$ is commutative, while $\oplus$ is not. In case both $\odot$ and $\oplus$ are used to describe a complex pattern, $\odot$ has precedence over $\oplus$, i.e.,

$$\mathcal{P}_1 \odot \mathcal{P}_2 \odot \mathcal{P}_3 \oplus \mathcal{P}_4 \odot \mathcal{P}_5 \oplus \mathcal{P}_6 \;\equiv\; ((\mathcal{P}_1 \odot \mathcal{P}_2 \odot \mathcal{P}_3) \oplus (\mathcal{P}_4 \odot \mathcal{P}_5) \oplus \mathcal{P}_6).$$

We use bracketing to overrule these assumptions or to avoid ambiguity. Further, we use the following notation to simplify complex terms where necessary and appropriate:

$$\circledcirc \in \{\oplus, \odot\}: \quad \mathcal{P}_1 \circledcirc \ldots \circledcirc \mathcal{P}_p \;\equiv\; \circledcirc(P_1, \ldots, \mathcal{P}_p) \;\equiv\; \circledcirc|_{q=1}^{p}(\mathcal{P}_q).$$

Table 2 gives some examples how to describe the access patterns of some typical database algorithms as compound patterns. For convenience, some re-occurring compound access patterns are assigned a new name.

Our hypothesis is, that we only need to provide an access pattern description as depicted in Table 2 for each operation we want to model. The actual cost function can then be created automatically, provided we know the cost functions for the basic patterns, and the rules how to combine them. To verify this hypothesis, we will now first estimate the cache miss rates of the basic access patterns and then derive rules how to calculate the cache miss rates of compound access patterns.

4. Deriving Cost Functions

In this section, N depicts the number of cache levels and i iterates over all levels: $i \in \{1, \ldots, N\}$. For better readability, we will omit the index i wherever we do not refer to a specific cache level, but rather to all or any.

algorithm	pattern description		name
$W \leftarrow \mathit{select}(U)$	$\mathsf{s\_trav}(U) \odot \mathsf{s\_trav}(W)$		
$W \leftarrow \mathit{nested\_loop\_join}(U,V)$	$\mathsf{s\_trav}(U) \odot \mathsf{rs\_trav}(U.n, \mathtt{uni}, V) \odot \mathsf{s\_trav}(W)$	$=:$	$\mathsf{nl\_join}(U,V,W)$
$W \leftarrow \mathit{zick\_zack\_join}(U,V)$	$\mathsf{s\_trav}(U) \odot \mathsf{rs\_trav}(U.n, \mathtt{bi}, V) \odot \mathsf{s\_trav}(W)$		
$H \leftarrow \mathit{hash\_build}(V)$	$\mathsf{s\_trav}(V) \odot \mathsf{r\_trav}(H)$	$=:$	$\mathsf{build\_hash}(V,H)$
$W \leftarrow \mathit{hash\_probe}(U,H)$	$\mathsf{s\_trav}(U) \odot \mathsf{r\_acc}(U.n, H) \odot \mathsf{s\_trav}(W)$	$=:$	$\mathsf{probe\_hash}(U,H,W)$
$W \leftarrow \mathit{hash\_join}(U,V)$	$\mathsf{build\_hash}(V,H) \oplus \mathsf{probe\_hash}(U,H,W)$	$=:$	$\mathsf{h\_join}(U,V,W)$
$\{U_j\}\vert_{j=1}^{m} \leftarrow \mathit{cluster}(U,m)$	$\mathsf{s\_trav}(U) \odot \mathsf{nest}(\{U_j\}\vert_{j=1}^{m}, m, \mathsf{s\_trav}(U_j), \mathtt{ran})$	$=:$	$\mathsf{part}(U, m, \{U_j\}\vert_{j=1}^{m})$
$W \leftarrow \mathit{part\_nl\_join}(U,V,m)$	$\mathsf{part}(U, m, \{U_j\}\vert_{j=1}^{m}) \oplus \mathsf{part}(V, m, \{V_j\}\vert_{j=1}^{m})$ $\oplus\, \mathsf{nl\_join}(U_1, V_1, W_1) \oplus \ldots \oplus \mathsf{nl\_join}(U_m, V_m, W_m)$		
$W \leftarrow \mathit{part\_h\_join}(U,V,m)$	$\mathsf{part}(U, m, \{U_j\}\vert_{j=1}^{m}) \oplus \mathsf{part}(V, m, \{V_j\}\vert_{j=1}^{m})$ $\oplus\, \mathsf{h\_join}(U_1, V_1, W_1) \oplus \ldots \oplus \mathsf{h\_join}(U_m, V_m, W_m)$		

Table 2: Sample Data Access Patterns

4.1 Preliminaries

For each basic pattern, we need to estimate both sequential and random cache misses for each cache level. Given an access pattern $\mathcal{P} \in \mathbb{P}$, we describe the number of misses per cache level as pair

$$\vec{\mathbf{M}}_i(\mathcal{P}) = \langle \mathbf{M}^{\mathtt{s}}_i(\mathcal{P}), \mathbf{M}^{\mathtt{r}}_i(\mathcal{P}) \rangle \in \mathbb{N} \times \mathbb{N} \tag{4.1}$$

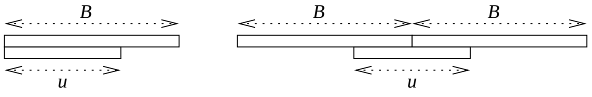
containing the number of sequential and random cache misses. Obviously, the random patterns cause only random misses, but no sequential misses. Consequently, we always set

$$\mathbf{M}^{\mathtt{s}}_i(\mathcal{P}) = 0 \qquad \text{for random patterns.}$$

Sequential traversals can achieve sequential latency (i.e., exploit full excess bandwidth), only if all the requirements listed in Section 2.2 are fulfilled. Sequential access is fulfilled by definition. The hardware requirements (non-blocking caches and super-scalar CPUs allowing speculative execution) are covered by the results of our calibration tool. In case these properties are not given, sequential latency will be the same as random latency. However, the pure existence of these hardware features is not sufficient to achieve sequential latency. Rather, the implementation needs to be able to exploit these features. Data dependencies in the code may keep the CPU from issuing multiple memory requests concurrently. It is not possible to deduce this information only from the algorithm without knowing the actual implementation. But even without data dependencies, multiple concurrent memory requests may hit the same cache line. In case the number of concurrent hits to a single cache line is lower than the maximal number of outstanding memory references allowed by the CPU, only one cache line is loaded at a time.[5] Though we can say how many subsequent references hit the same cache line (see below), we do not know how many outstanding memory references the CPU can handle without stalling.[6] Hence, it is not possible to automatically guess, whether a sequential traversal can achieve sequential latency or not. For this reason, we offer two variants of s_trav and rs_trav. $\mathsf{s\_trav}^{\mathtt{s}}$ and $\mathsf{rs\_trav}^{\mathtt{s}}$ assume a scenario that can achieve sequential latency while $\mathsf{s\_trav}^{\mathtt{r}}$ and $\mathsf{rs\_trav}^{\mathtt{r}}$ do not. In the first case, we get only sequential but no random misses. The second case gives only random but no sequential misses. The actual number of misses is equal in both cases. Unless we need to explicitly distinguish between both variants, we will use $\mathsf{s\_trav}^{x}$ respectively $\mathsf{rs\_trav}^{x}$ to refer to both ($x \in \{\mathtt{s}, \mathtt{r}\}$). When describing the access pattern of a certain algorithm, we will use the variant that fits to the actual code.

[5] For a more detailed discussion, we refer the interested reader to [MBK02].

[6] Our calibration results can only indicate, whether the CPU can handle outstanding memory references without stalling, but not how many it can handle concurrently.



a) 1 cache miss b) 2 cache misses

Figure 4: Impact of Alignment on the Number of Cache Misses

4.2 Single Sequential Traversal

Be R a data region and $\mathcal{P} = \mathsf{s\_trav}^x(R,u)$ a sequential traversal over R. As mentioned above, we have

$$\mathbf{M}^{\mathsf{r}}_i(\mathsf{s\_trav}^{\mathsf{s}}(R,u)) = 0 \quad \text{and} \quad \mathbf{M}^{\mathsf{s}}_i(\mathsf{s\_trav}^{\mathsf{r}}(R,u)) = 0.$$

To calculate $\mathbf{M}^x_i(\mathcal{P})$, we distinguish two cases: $R.w - u < B$ and $R.w - u \geq B$.

Case $R.w - u < B$**.** In this case, the gap between two adjacent accesses that is not touched at all is smaller than a single cache line. Hence, the cache line containing this gap is loaded to serve at least one of the two adjacent accesses. Thus, during a sweep over R with $R.w - u < B$ all cache lines coved by R have to be loaded, i.e.,

$$\mathbf{M}^x_i(\mathsf{s\_trav}^x(R,u)) = |R|_{B_i}. \tag{4.2}$$

Case $R.w - u \geq B$**.** In this case, the gap between two adjacent accesses that is not touched at all spans at least a complete cache line. Hence, not all cache lines coved by R have to be loaded during a sweep over R with $R.w - u \geq B$. Further, no access can benefit from a cache line already loaded by a previous access to another spot. Thus, each access to an item in R requires at least $\lceil \frac{u}{B} \rceil$ cache lines to be loaded. We get

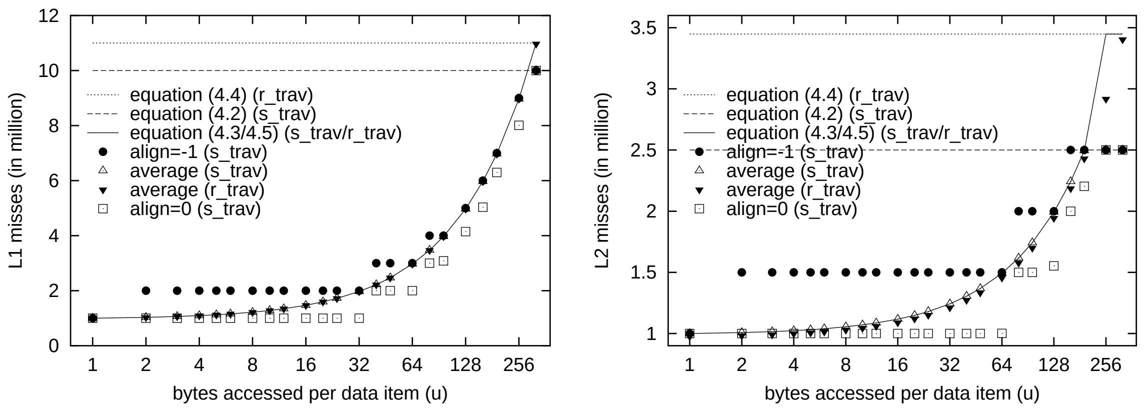
$$\mathbf{M}^x_i(\mathsf{s\_trav}^x(R,u)) \geq R.n \cdot \left\lceil \frac{u}{B_i} \right\rceil.$$

However, with $u > 1$ it may happen that — depending on the alignment of u within a cache line — one additional cache line has to be loaded per access. Figure 4 depicts such a scenario.

The actual alignment of each u in a sweep is determined by two parameters. First, it of course depends on the alignment of the first item in R, i.e., the alignment of R itself. Assuming a 1 byte access granularity, R can be aligned on B places within a cache line. Second, $R.w$ determines whether all items in R are aligned equally, or whether their alignment changes throughout R. In case $R.w$ is a multiple of B, all items in R are equally aligned as the first one. Otherwise, the alignment varies throughout R, but picking only $\frac{B}{\gcd\{B,R.w\}}$ out of the B theoretically possible places. As we don't know anything about the alignment of R, there is no way of reasonably exploiting the information we just learned about the impact of $R.w$ on the alignment shift. Hence, all we can do is assuming that all B possibilities occur equally often. All we need to do now, is count how many of these B possibilities yield an additional cache miss. For convenience, we define

$$x \mathbin{\widehat{\text{mod}}} y = \begin{cases} y, & \text{if } x \bmod y = 0, \\ x \bmod y, & \text{else.} \end{cases}$$

When u is aligned on the first position (i.e., the first byte) in a cache line, no more than $\lceil \frac{u}{B} \rceil$ cache lines have to be loaded to access whole u. In this case, the last $B - (u \mathbin{\widehat{\text{mod}}} B)$ bytes in the last cache line loaded for u are not used by u. Hence, shifting u's position by up to $B - (u \mathbin{\widehat{\text{mod}}} B)$ also doesn't require any additional



a) $\mathbf{M}_1(\mathsf{s\_trav}(R,u))$, $\mathbf{M}_1(\mathsf{r\_trav}(R,u))$ b) $\mathbf{M}_2(\mathsf{s\_trav}(R,u))$, $\mathbf{M}_2(\mathsf{r\_trav}(R,u))$

Figure 5: Impact of u and its Alignment on Cache Misses ($R.n = 1{,}000{,}000$, $R.w = 320\,\mathrm{B}$, $B_1 = 32\,\mathrm{B}$, $B_2 = 128\,\mathrm{B}$)

cache miss. Only the remaining

$$\begin{aligned}
&B - 1 - (B - (u \;\widehat{\bmod}\; B)) \\
&= (u \;\widehat{\bmod}\; B) - 1 \\
&= \begin{cases} B-1, & \text{if } u \bmod B = 0 \\ (u \bmod B) - 1, & \text{else} \end{cases} \\
&\overset{u \geq 0}{=} (u-1) \bmod B
\end{aligned}$$

positions will yield one additional cache miss. Putting all pieces together, we get:

$$\mathbf{M}_i^x(\mathsf{s\_trav}^x(R,u)) = R.n \cdot \left(\left\lceil \frac{u}{B_i} \right\rceil + \frac{(u-1) \bmod B_i}{B_i} \right). \tag{4.3}$$

Figure 5 demonstrates the impact of u on the number of cache misses. The points show the number of cache misses measured with various alignments. ”$\mathsf{align} = 0$” and ”$\mathsf{align} = -1$” make up the two extreme cases. In the first case, u is aligned on the first byte of a cache line; in the second case, u starts on the last byte of a cache line. ”$\mathsf{average}$” depicts the average over all possible alignments. The dotted curve and the dashed curve represent Equations (4.4) and (4.2), respectively, which ignore u and assume that all $R.w$ bytes of each item are touched. The solid curve represents the identical Equations (4.3) and (4.5) which consider u. The graphs show, that u has a significant impact on the number of cache misses, and that our formulas correctly predict the average impact.

4.3 Single Random Traversal

Be R a data region and $\mathcal{P} = \mathsf{r\_trav}(R,u)$ a random traversal over R. As mentioned above, we have

$$\mathbf{M}_i^{\mathsf{s}}(\mathsf{r\_trav}(R,u)) = 0.$$

Like with sequential traversal, we will distinguish two cases: $R.w - u < B$ and $R.w - u \geq B$.

Case $R.w - u < B$**.** With the untouched gaps being smaller than cache line size, again all cache lines coved by R have to be accessed. Hence, $\mathbf{M}^{\mathbf{r}}(\mathcal{P}) \geq |R|_B$. But due to the random access pattern, two locally adjacent accesses are not temporally adjacent. Thus, if $||R||$ exceeds the cache size, a cache line that serves two or more (locally adjacent) accesses may be replaced by another cache line before all accesses that require it actually took place. This in turn causes an additional cache miss, once the original cache line is accessed again. Of course, such additional cache misses only occur, once the cache capacity is exceeded, i.e., after $\min\{\#_i, |C_i|_{R.w}\}$ spots have been accessed. The probability that a cache line is removed from the cache although it will be used for another access increases with the size of R. In the worst case, each access causes an additional cache miss. Hence, we get

$$\mathbf{M}_i^{\mathbf{r}}(\mathsf{r\_trav}(R,u)) = |R|_{B_i} + (R.n - \min\{\#_i, |C_i|_{R.w}\}) \cdot \left(1 - \min\left\{1, \frac{C_i}{||R||}\right\}\right). \tag{4.4}$$

Case $R.w - u \geq B$**.** Each spot is touched exactly once, and as adjacent accesses cannot benefit from previously loaded cache lines, we get the same formula for sequential access:

$$\mathbf{M}_i^{\mathbf{r}}(\mathsf{r\_trav}(R,u)) = R.n \cdot \left(\left\lceil \frac{u}{B_i} \right\rceil + \frac{(u-1) \bmod B_i}{B_i}\right). \tag{4.5}$$

4.4 Discussion

Comparing the final formulae for sequential and random traversals, we can derive the following relationships and invariants. Figure 6 visualizes some of the effects. The points represent the measure cache misses, while the lines represent the estimations of our formulas.

$$\begin{aligned} R.w - u < B_i \ \wedge\ ||R|| \leq C_i \quad &\overset{(4.2,4.4)}{\Rightarrow} \quad \mathbf{M}_i^{x}(\mathsf{s\_trav}^{x}(R,u)) = \mathbf{M}_i^{\mathbf{r}}(\mathsf{r\_trav}(R,u)); \\ R.w - u < B_i \ \wedge\ ||R|| > C_i \quad &\overset{(4.2,4.4)}{\Rightarrow} \quad \mathbf{M}_i^{x}(\mathsf{s\_trav}^{x}(R,u)) < \mathbf{M}_i^{\mathbf{r}}(\mathsf{r\_trav}(R,u)). \end{aligned}$$

With untouched gaps smaller than cache lines, random traversals cause as many misses as sequential traversals as long as R fits in the cache, but more, if R exceeds the cache (cf. Figure 6a vs. 6c & 6b vs. 6d).

$$R.w - u \geq B_i \quad \overset{(4.3,4.5)}{\Rightarrow} \quad \mathbf{M}_i^{x}(\mathsf{s\_trav}^{x}(R,u)) = \mathbf{M}_i^{\mathbf{r}}(\mathsf{r\_trav}(R,u)).$$

With untouched gaps larger than cache lines, random traversals cause as many misses as sequential traversals.

$$\begin{aligned} R.w - u < B_i \quad \overset{(4.2)}{\Rightarrow} \quad &\mathbf{M}_i^{x}(\mathsf{s\_trav}^{x}(R,u)) = \mathbf{M}_i^{x}(\mathsf{s\_trav}^{x}(R',u')) \\ &\forall R', u' \text{ with } ||R'|| = ||R|| \ \wedge\ R'.w - u' < B_i. \end{aligned}$$

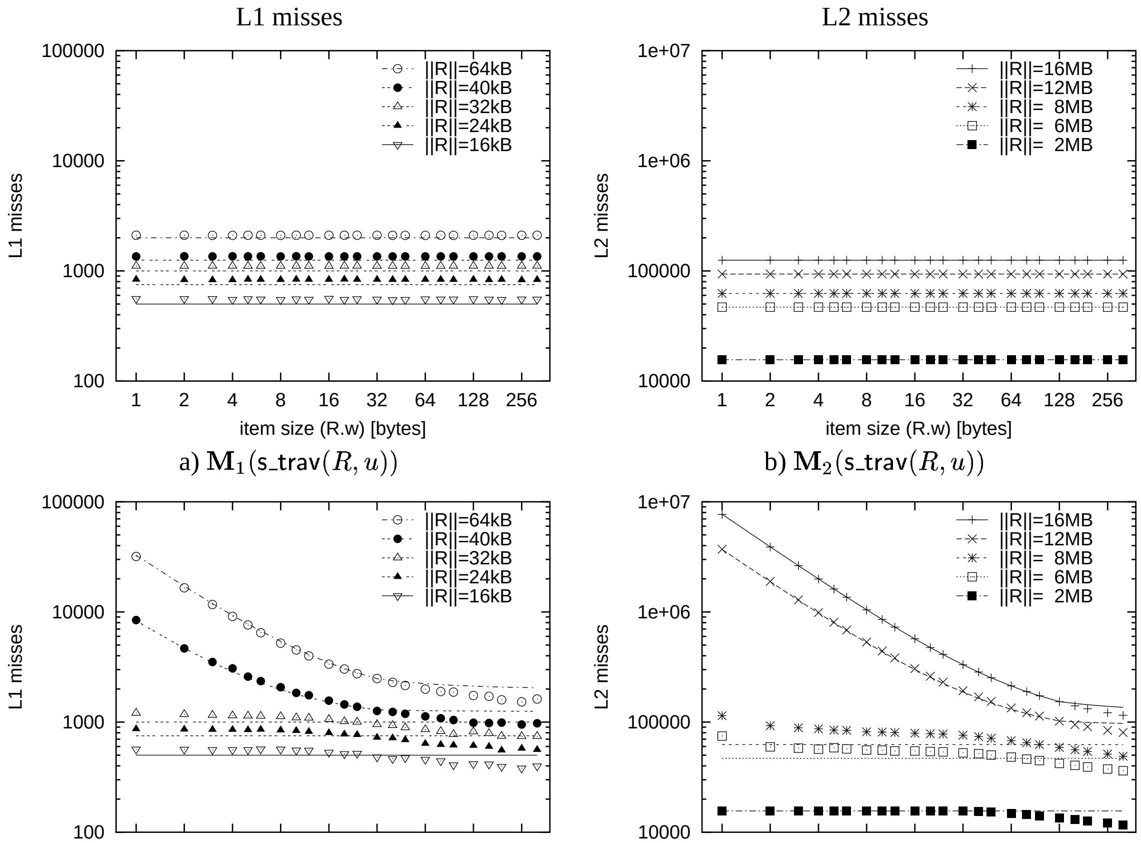
With untouched gaps smaller than cache lines, sequential traversals depend only on the size of R, but are invariant to varying item size (and hence number of items) and bytes touched per item (cf. Figure 6a & 6b).

$$\begin{aligned} R.w - u < B_i \ \wedge\ ||R|| \leq C_i \quad \overset{(4.4)}{\Rightarrow} \quad &\mathbf{M}_i^{\mathbf{r}}(\mathsf{r\_trav}(R,u)) = \mathbf{M}_i^{\mathbf{r}}(\mathsf{r\_trav}(R',u')) \\ &\forall R', u' \text{ with } ||R'|| = ||R|| \ \wedge\ R'.w - u' < B_i; \\ R.w - u < B_i \ \wedge\ ||R|| > C_i \quad \overset{(4.4)}{\Rightarrow} \quad &\mathbf{M}_i^{\mathbf{r}}(\mathsf{r\_trav}(R,u)) = \mathbf{M}_i^{\mathbf{r}}(\mathsf{r\_trav}(R,u')) \\ &\forall u' \text{ with } R.w - u' < B_i. \end{aligned}$$

For random traversals, the invariance to item size holds only if R entirely fits in the cache (cf. Figure 6c & 6d).

$$\begin{aligned} R.w - u \geq B_i \quad \overset{(4.3/4.5)}{\Rightarrow} \quad &\vec{\mathbf{M}}_i(\mathsf{T}(R,u)) = \vec{\mathbf{M}}_i(\mathsf{T}(R',u)) \\ &\forall R' \text{ with } R'.n = R.n \ \wedge\ R'.w - u \geq B_i, \\ &\mathsf{T} \in \{\mathsf{s\_trav}^{x}, \mathsf{r\_trav}\}. \end{aligned}$$

With untouched gaps larger than cache lines, the number of misses of all traversals depend only on the number of items accessed and the number of bytes touched per item.



a) $\mathbf{M}_1(\mathsf{s\_trav}(R,u))$

b) $\mathbf{M}_2(\mathsf{s\_trav}(R,u))$

c) $\mathbf{M}_1(\mathsf{r\_trav}(R,u))$

d) $\mathbf{M}_2(\mathsf{r\_trav}(R,u))$

Figure 6: Impact of $||R||$ and $R.w$ on Cache Misses ($u = R.w$, $C_1 = 32\,\mathrm{kB}$, $B_1 = 32\,\mathrm{B}$, $C_2 = 8\,\mathrm{MB}$, $B_2 = 128\,\mathrm{B}$)

4.5 Repetitive Traversals

With repetitive traversals, cache re-usage comes into play. We assume initially empty caches. Hence, the first traversal requires as many cache misses as estimated above. But the subsequent traversals may benefit from the data already present in the cache after the first access. We will analyze this in detail for both sequential and random traversals.

4.5.1 Repetitive Sequential Traversal Be R a data region, $\mathcal{P} = \mathsf{rs\_trav}^x(r,d,R,u)$ a repetitive sequential traversal over R, and $\mathcal{P}' = \mathsf{s\_trav}^x(R,u)$ a single sequential traversal over R. Two parameters determine the caching behavior of a repetitive sequential traversal: the number $\mathbf{M}^x(\mathcal{P}')$ of cache lines touched during the first traversal and the direction d in which subsequent traversals sweep over R.

In case $\mathbf{M}^x(\mathcal{P}')$ is smaller than the total number of available cache lines, only the first traversal causes cache misses, loading all required data. All $r-1$ subsequent traversals then just access the cache, causing no further cache misses.

In case $\mathbf{M}^x(\mathcal{P}')$ exceeds the number of available cache lines, the end of a traversal pushes the data read at the begin of the traversal out of the cache. If the next traversal than again starts at the begin of R, it cannot benefit from any data in the cache. Hence, with $d = \texttt{uni}$, each sweep causes the full amount of cache misses. Only if a subsequent sweep starts where the previous one stopped, i.e., it traverses R in the opposite direction as its predecessor, it can benefit from the data stored in the cache. Thus, with $d = \mathtt{bi}$, only the first sweep causes the full amount of cache misses. The $r - 1$ remaining sweeps cause cache misses only for the fraction of R that does not fit into the cache.

In total, we get ($\mathcal{P}' = \mathsf{s\_trav}^x(R, u)$)

$$\mathbf{M}_i^x(\mathsf{rs\_trav}^x(r, d, R, u)) \\ = \begin{cases} \mathbf{M}_i^x(\mathcal{P}'), & \text{if} \quad \mathbf{M}_i^x(\mathcal{P}') \leq \#_i \\ r \cdot \mathbf{M}_i^x(\mathcal{P}'), & \text{if} \quad \mathbf{M}_i^x(\mathcal{P}') > \#_i \wedge d = \mathtt{uni} \\ \mathbf{M}_i^x(\mathcal{P}') + (r-1) \cdot (\mathbf{M}_i^x(\mathcal{P}') - \#_i), & \text{if} \quad \mathbf{M}_i^x(\mathcal{P}') > \#_i \wedge d = \mathtt{bi}. \end{cases} \tag{4.6}$$

4.5.2 Repetitive Random Traversal Be R a data region, $\mathcal{P} = \mathsf{rr\_trav}(r, R, u)$ a repetitive random traversal over R, and $\mathcal{P}' = \mathsf{r\_trav}(R, u)$ a single random traversal over R. With random memory access, d is not defined, hence, we need to consider only $\mathbf{M}^{\mathtt{r}}(\mathcal{P}')$ to determine to which extend repetitive accesses can benefit from cached data.

When $\mathbf{M}^{\mathtt{r}}(\mathcal{P}')$ is smaller than the number of available cache lines, we get the same effect as above. Only the first sweep causes cache misses, loading all required data. All $r - 1$ subsequent sweeps then just access the cache, causing no further cache misses.

In case $\mathbf{M}^{\mathtt{r}}(\mathcal{P}')$ exceeds the number of available cache lines, the most recently accessed data remains in the cache at the end of a sweep. Hence, there is a certain probability that the first accesses of the following sweep might re-use (some of) these $\#$ cache lines. This probability decreases as $\mathbf{M}^{\mathtt{r}}(\mathcal{P}')$ increases. We estimate the probability with $\#/\mathbf{M}^{\mathtt{r}}(\mathcal{P}')$.

Analogously to the sequential case, we get ($\mathcal{P}' = \mathsf{r\_trav}(R, u)$)

$$\mathbf{M}_i^{\mathtt{r}}(\mathsf{rr\_trav}(r, R, u)) \\ = \begin{cases} \mathbf{M}_i^{\mathtt{r}}(\mathcal{P}'), & \text{if} \quad \mathbf{M}_i^{\mathtt{r}}(\mathcal{P}') \leq \#_i \\ \mathbf{M}_i^{\mathtt{r}}(\mathcal{P}') + (r-1) \cdot \left(\mathbf{M}_i^{\mathtt{r}}(\mathcal{P}') - \dfrac{\#_i}{\mathbf{M}_i^{\mathtt{r}}(\mathcal{P}')} \cdot \#_i \right), & \text{if} \quad \mathbf{M}_i^{\mathtt{r}}(\mathcal{P}') > \#_i. \end{cases} \tag{4.7}$$

4.6 Random Access

Be R a data region and $\mathcal{P} = \mathsf{r\_acc}(r, R, u)$ a random access pattern on R. As in Section 4.3, we have

$$\mathbf{M}_i^{\mathsf{s}}(\mathsf{r\_acc}(R, u)) = 0.$$

In contrary to a single random traversal, where each data item of R is touched exactly once, we do not know exactly, how many distinct data items are actually touched with random access. However, knowing that there are r independent random accesses to the $R.n$ data items in R, we can estimate the average/expected number $\mathbf{I}$ of distinct data items that are indeed touched. Be E the number of all different outcomes of picking r times one of the $R.n$ data items allowing multiple accesses to each data item. Further be E_j the number of outcomes containing exactly $1 \leq j \leq \min\{r, R.n\}$ distinct data items. If we respect ordering, all outcomes are equally likely to occur, hence, we have

$$\mathbf{I}(\mathsf{r\_acc}(r, R, u)) = \frac{\sum_{j=1}^{\min\{r, R.n\}} E_j(r, R.n) \cdot j}{E(r, R.n)} \qquad \text{with} \qquad \sum_{j=1}^{\min\{r, R.n\}} E_j(r, R.n) = E(r, R.n).$$

Calculating E is straight forward:

$$E(r, R.n) = R.n^r.$$

Calculating E_j turns out to be a bit more difficult. Be

$$\binom{x}{y} = \frac{x!}{(x-y)! \cdot y!}$$

the *binomial coefficient*, i.e., the number of ways of picking y unordered outcomes from x possibilities. Further be

$$\left\{ {x \atop y} \right\} = \frac{1}{y!} \cdot \sum_{j=0}^{y-1} (-1)^j \cdot \binom{y}{j} \cdot (y-j)^x$$

the *Stirling number of second kind*, i.e., the number of ways of partitioning a set of x elements into y nonempty sets. Then, we have

$$E_j(r, R.n) = \binom{R.n}{j} \cdot \left\{ {r \atop j} \right\} \cdot j!.$$

First of all, there are $\binom{R.n}{j}$ ways to choose j distinct data items from the available $R.n$ data items. Then, there are $\left\{ {r \atop j} \right\}$ ways to partition the r access into j groups, one for each distinct data item. Finally, we have to consider all $j!$ permutations to get equally likely outcomes.

Knowing the number $\mathbf{I}$ of distinct data items that are touched by $\mathsf{r\_acc}(r, R, u)$ on average, we can now calculate the number $\mathbf{C}$ of distinct cache lines touched. Again, we distinguish two cases, depending on the size of the (minimal) untouched gaps between two adjacent accesses.

Case $R.w - u \geq B$. With the (minimal) untouched gaps larger than cache line size, no cache lines is used by more than one data item. Following the discussion in Section 4.2, we get

$$\mathbf{C}_i(\mathsf{r\_acc}(r, R, u)) = \mathbf{I}(\mathsf{r\_acc}(r, R, u)) \cdot \left(\left\lceil \frac{u}{B_i} \right\rceil + \frac{(u-1) \bmod B_i}{B_i} \right).$$

Case $R.w - u \geq B$. With the (minimal) untouched gaps smaller than cache line size, (some) cache lines might be used by more than one data item. In case all $\mathbf{I}$ touched data items are pair-wise adjacent, we get

$$\check{\mathbf{C}}_i(\mathsf{r\_acc}(r, R, u)) = \left\lceil \frac{\mathbf{I}(\mathsf{r\_acc}(r, R, u)) \cdot R.w}{B_i} \right\rceil.$$

However, if $\mathbf{I} \ll R.n$, the actual untouched gap might still be larger than cache line size, hence

$$\hat{\mathbf{C}}_i(\mathsf{r\_acc}(r, R, u)) = \min \left\{ \mathbf{I}(\mathsf{r\_acc}(r, R, u)) \cdot \left(\left\lceil \frac{u}{B_i} \right\rceil + \frac{(u-1) \bmod B_i}{B_i} \right), |R|_{B_i} \right\}.$$

$\check{\mathbf{C}}$ is more likely with large $\mathbf{I}$, while $\hat{\mathbf{C}}$ is more likely with small $\mathbf{I}$. Hence, we calculate the average $\mathbf{C}$ as a linear combination of $\check{\mathbf{C}}$ and $\hat{\mathbf{C}}$:

$$\mathbf{C}_i(\mathsf{r\_acc}(r, R, u)) = \frac{\mathbf{I}(\mathsf{r\_acc}(r, R, u))}{R.n} \cdot \check{\mathbf{C}}_i(\mathsf{r\_acc}(r, R, u)) + \left(1 - \frac{\mathbf{I}(\mathsf{r\_acc}(r, R, u))}{R.n}\right) \cdot \hat{\mathbf{C}}_i(\mathsf{r\_acc}(r, R, u)).$$

Knowing the number $\mathbf{C}$ of distinct cache lines touched, we can finally calculate the number of cache misses. With r accesses spread over $\mathbf{I}$ distinct data items, each item is touched $r/\mathbf{I}$ times on average. Analogously to Equation (4.7), we get ($\mathcal{P} = \mathsf{r\_acc}(r, R, u)$)

$$\mathbf{M}_i^{\mathbf{r}}(\mathsf{r\_acc}(r, R, u)) = \begin{cases} \mathbf{C}_i(\mathcal{P}), & \text{if} \quad \mathbf{C}_i(\mathcal{P}) \leq \#_i \\ \mathbf{C}_i(\mathcal{P}) + \left(\dfrac{r}{\mathbf{I}(\mathcal{P})} - 1 \right) \cdot \left(\mathbf{C}_i(\mathcal{P}) - \dfrac{\#_i}{\mathbf{C}_i(\mathcal{P})} \cdot \#_i \right), & \text{if} \quad \mathbf{C}_i(\mathcal{P}) > \#_i. \end{cases} \tag{4.8}$$

4.7 Interleaved Multi-Cursor Access

Be $R = \{R_j\}|_{j=1}^{m}$ a data region divided into $m \leq R.n$ sub-regions R_j with

$$R_j.w = R.w \qquad \text{and} \qquad k = R_j.n = \frac{R.n}{m}. \tag{$*$}$$

Further be $\mathcal{Q} = \mathsf{nest}(R, m, \mathsf{T}([r,]R_j, u), O, D)$ with $\mathsf{T} \in \{\mathsf{s\_trav}^x, \mathsf{r\_trav}, \mathsf{r\_acc}\}$ an interleaved multi-cursor access. We inspect local random access ($\mathsf{T} \in \{\mathsf{r\_acc}, \mathsf{r\_trav}\}$) and local sequential access ($\mathsf{T} = \mathsf{s\_trav}^x$) separately.

4.7.1 Local Random Access With $\mathsf{T} \in \{\mathsf{r\_acc}, \mathsf{r\_trav}\}$, $\mathcal{Q}$ behaves like a single traversal $\mathcal{Q}' = \mathsf{T}'([m \cdot r,]R, u)$. For $k = 1$ (i.e., $m = R.n$), the new order is the original global order, otherwise, it is the original local order, i.e., we get

$$\mathsf{T}' = \begin{cases} \mathsf{s\_trav}^{x}, & \text{if} \quad k = 1 \ \wedge \ O = \mathtt{seq} \\ \mathsf{T}, & \text{else} \end{cases}$$

and consequently

$$\vec{\mathbf{M}}_i(\mathsf{nest}(R, m, \mathsf{T}([r,]R_j, u), O, D)) = \vec{\mathbf{M}}_i(\mathsf{T}'([m \cdot r,]R, u)).$$

4.7.2 Local Sequential Access For $\mathsf{T} = \mathsf{s\_trav}^{x}$, we distinguish three cases:

- $R.w - u > B$,
- $R.w - u \leq B \ \wedge \ m \cdot \lceil \frac{u}{B} \rceil \leq \#$,
- $R.w - u \leq B \ \wedge \ m \cdot \lceil \frac{u}{B} \rceil > \#$.

Case $R.w - u > B$**.** In this case, $\mathcal{Q}$ means $k = R_j.n$ times traversing across all R_j in order O. Hence, each traversal performs m accesses (one to each R_j). The distance between adjacent accesses within each traversal is $||R_j|| = k \cdot R.w$. We describe these traversals by $\mathsf{T}'(R', u)$ where R' is a data region with

$$R'.n = m \quad \text{and} \quad R'.w = ||R_j||, \tag{†}$$

and

$$\mathsf{T}' = \begin{cases} \mathsf{r\_trav}, & \text{if} \quad O = \mathtt{ran} \\ \mathsf{T}, & \text{else.} \end{cases}$$

As the non-touched gap between adjacent accesses within each R_j is larger than a cache line ($R.w - u > B$), no cache line is shared by two or more accesses. Thus, the total number of cache misses is the sum of the cache misses caused by the k traversals:

$$\begin{aligned} &\vec{\mathbf{M}}_i(\mathsf{nest}(R, m, \mathsf{T}(R_j, u), O, D)) \\ &\quad = \quad \sum_{h=1}^{k} \vec{\mathbf{M}}_i(\mathsf{T}'(R', u)) \\ &\quad = \quad k \cdot \vec{\mathbf{M}}_i(\mathsf{T}'(R', u)) \\ &\quad \overset{(4.3/4.5)}{=} \quad k \cdot R'.n \cdot \left(\left\lceil \frac{u}{B_i} \right\rceil + \frac{(u-1) \bmod B_i}{B_i} \right) \\ &\quad \overset{(4.3/4.5,*,\dagger)}{=} \quad \vec{\mathbf{M}}_i(\mathsf{T}'(R, u)). \end{aligned}$$

Case $R.w - u \leq B \quad \wedge \quad m \cdot \lceil \frac{u}{B} \rceil \leq \#$**.** With the non-touched gaps being smaller than cache line size ($R.w - u \leq B$), adjacent accesses within each R_j might shared a cache line, and hence benefit from previous accesses. With one traversal across all R_j touching less cache lines than there are in total ($m \cdot \lceil \frac{u}{B} \rceil \leq \#$), the subsequent traversal does not have to reload the shared cache lines. Hence, the total number of cache misses is just the sum of all local patterns. Though these are sequential, a global random pattern will avoid sequential latency. We take this into account when defining T' and get

$$\begin{aligned}
&\vec{\mathbf{M}}_i(\mathsf{nest}(R, m, \mathsf{T}(R_j, u), O, D))\\
&\quad = \sum_{j=1}^{m} \vec{\mathbf{M}}_i(\mathsf{T}'(R_j, u))\\
&\quad = m \cdot \vec{\mathbf{M}}_i(\mathsf{T}'(R_j, u))\\
&\quad \overset{(4.2,*)}{=} \vec{\mathbf{M}}_i(\mathsf{T}'(R, u))
\end{aligned}$$

with

$$\mathsf{T}' = \begin{cases} \mathsf{s\_trav}^{r}, & \text{if} \quad O = \mathtt{ran}\\ \mathsf{T}, & \text{else.} \end{cases}$$

Case $R.w - u \leq B \quad \wedge \quad m \cdot \left\lceil \frac{u}{B} \right\rceil > \#$**.** With one traversal across all R_j touching more cache lines than there are in total ($m \cdot \left\lceil \frac{u}{B} \right\rceil > \#$), only $h = \# / \left\lceil \frac{u}{B} \right\rceil < m$ of the m shared cache lines remain in the cache for potential re-use. The number h' of cache lines that is actually re-used depends on O and D and is calculated similarly as for the repetitive traversals in Section 4.5:

$$h'_i = \begin{cases} 0, & \text{if } O = \mathtt{seq} \wedge D = \mathtt{uni}\\ h_i, & \text{if } O = \mathtt{seq} \wedge D = \mathtt{bi}\\ \dfrac{h_i}{m} \cdot h_i, & \text{if } O = \mathtt{ran} \end{cases}$$

$$h_i = \#_i / \left\lceil \frac{u}{B_i} \right\rceil .$$

Hence, the total number of cache misses is the same as in the previous case, plus the $m - h$ cache lines that have to be reloaded during all but the first traversal. These additional misses cause random latency, i.e., we get

$$\vec{\mathbf{M}}_i(\mathsf{nest}(R, m, \mathsf{T}(R_j, u), O, D)) = \vec{\mathbf{M}}_i(\mathsf{T}'(R, u)) + \vec{\mathbf{X}}_i$$

with

$$\vec{\mathbf{X}}_i = \langle 0, (k-1) \cdot (m - h'_i) \rangle \qquad (\ddagger)$$

and T' as before.

4.7.3 Summary Gathering the results from all the different cases discussed above, we get

$$\begin{aligned}
&\vec{\mathbf{M}}_i(\mathsf{nest}(R, m, \mathsf{T}([r,]R_j, u), O, D))\\
&\quad = \begin{cases} \vec{\mathbf{M}}_i(\mathsf{T}'([m \cdot r,]R, u)) + \vec{\mathbf{X}}_i, & \text{if} \quad \mathsf{T} = \mathsf{s\_trav}^{x} \wedge R.w - u < B_i \wedge m \cdot \left\lceil \frac{u}{B_i} \right\rceil > \#_i\\ \vec{\mathbf{M}}_i(\mathsf{T}'([m \cdot r,]R, u)), & \text{else} \end{cases} \qquad (4.9)
\end{aligned}$$

with $\vec{\mathbf{X}}_i$ as in (‡) and

$$\mathsf{T}' = \begin{cases} \mathsf{s\_trav}^{x}, & \text{if} \quad \mathsf{T} \in \{\mathsf{r\_acc}, \mathsf{r\_trav}\} \wedge O = \mathtt{seq} \wedge k = 1\\ \mathsf{r\_trav}, & \text{if} \quad \mathsf{T} = \mathsf{s\_trav}^{x} \wedge O = \mathtt{ran} \wedge R.w - u > B_i\\ \mathsf{s\_trav}^{r}, & \text{if} \quad \mathsf{T} = \mathsf{s\_trav}^{s} \wedge O = \mathtt{ran} \wedge R.w - u \leq B_i\\ \mathsf{T}, & \text{else.} \end{cases}$$

In other words, an interleaved multi-cursor access pattern causes at least as many cache misses as some simple traversal pattern on the same data region. However, it might cause random misses though the local pattern is expected to cause sequential misses. Further, if the cross-traversal requires more cache lines than available, additional random misses will occur.

5. Combining Cost Functions

Given the cache misses for basic patterns, we will now discuss how to derive the resulting cache misses of compound patterns. The major problem is to model cache interference that occur among the basic patterns.

5.1 Sequential Execution

Be $\mathcal{P}_1, \ldots, \mathcal{P}_p \in \mathbb{P}$ ($p > 1$) access patterns. $\oplus(\mathcal{P}_1, \ldots, \mathcal{P}_p)$ then denotes that $\mathcal{P}_{q+1}$ is executed after $\mathcal{P}_q$ is finished (cf. Sec. 3.3). Obviously, the patterns do not interfere in this case. Consequently, the resulting total number of cache misses is at most the sum of the cache misses of all p patterns. However, if two subsequent patterns operate on the same data region, the second might benefit from the data that the first one leaves in the cache. It depends on the cache size, the data sizes, and the characteristics of the individual patterns, how many cache misses may be saved this way.

To model this effect, we need to consider the contents or *state* of the caches. We describe the state of a cache as a set $\mathbf{S}$ of pairs $\langle R, \rho \rangle \in \mathbb{D} \times]0, 1]$, stating for each data region R the fraction ρ that is available in the cache. For convenience, we omit data regions that are not cached at all, i.e., those with $\rho = 0$. In order to appropriately consider the caches' initial states when calculating the cache misses of a pattern $\mathcal{P} = \mathsf{T}([..,]R[,..]) \in \mathbb{P}$, we define

$$\vec{\mathbf{M}}_i(\mathbf{S}_i, \mathcal{P}) = \begin{cases} \langle 0, 0 \rangle\,, & \text{if} \quad \langle R, 1 \rangle \in \mathbf{S}_i \\ \vec{\mathbf{M}}_i(\mathcal{P}) - \left\langle 0, \dfrac{\rho \cdot |R|_{B_i}}{\vec{\mathbf{M}}_i(\mathcal{P})} \cdot \rho \cdot |R|_{B_i} \right\rangle, & \text{if} \quad \mathsf{T} \in \{\mathsf{r\_trav}, \mathsf{rr\_trav}, \mathsf{r\_acc}\} \\ & \quad \wedge\ \exists \rho \in]0,1[\colon \langle R, \rho \rangle \in \mathbf{S}_i \\ \vec{\mathbf{M}}_i(\mathcal{P}) & \text{else} \end{cases} \tag{5.1}$$

with $\vec{\mathbf{M}}_i(\mathcal{P})$ as defined in Equations (4.2) through (4.9). In case R is already entirely available in the cache, no cache misses will occur during $\mathcal{P}$. In case only a fraction of R is available in the cache, there is a certain chance, that random patterns might (partially) benefit from this fraction. Sequential patterns, however, would only benefit if this fraction makes up the "head" of R. As we do not know whether this is true, we assume that sequential patterns can only benefit, if R is already entirely in the cache. For convenience, we write

$$\vec{\mathbf{M}}_i(\emptyset, \mathcal{P}) = \vec{\mathbf{M}}_i(\mathcal{P}) \qquad \forall \mathcal{P} \in \mathbb{P}.$$

Additionally, we need to know the caches' resulting states $\mathbf{S}(\mathcal{P})$ after a pattern $\mathcal{P}$ has been performed. For $\mathcal{P} = \mathsf{T}([..,]R[,..]) \in \mathbb{P}$, we define

$$\mathbf{S}_i(\mathcal{P}) = \left\{ \left\langle R, \min\left\{ \frac{C_i}{||R||}, 1 \right\} \right\rangle \right\}.$$

For compound patterns $\oplus(\mathcal{P}_1, \ldots, \mathcal{P}_p)$ with $\mathcal{P}_1, \ldots, \mathcal{P}_p \in \mathbb{P}, p > 1$, we define

$$\mathbf{S}_i(\oplus(\mathcal{P}_1, \ldots, \mathcal{P}_p)) = \mathbf{S}_i(\mathcal{P}_p).$$

Here, we assume that only that last data region (partially) remains in the cache. In case that R is smaller that the cache, (parts) of the previous data regions might also remain in the cache. However, we ignore this case here, and leave it for future research.

Equipped with these tools, we can finally calculate the number of cache misses that occur when executing patterns $\mathcal{P}_1, \ldots, \mathcal{P}_p \in \mathbb{P}, p > 1$ sequentially, given an initial cache state $\mathbf{S}$:

$$\vec{\mathbf{M}}_i(\mathbf{S}_i, \oplus(\mathcal{P}_1, \ldots, \mathcal{P}_p)) = \vec{\mathbf{M}}_i(\mathbf{S}_i, \mathcal{P}_1) + \sum_{q=2}^{p} \vec{\mathbf{M}}_i(\mathbf{S}_i(\mathcal{P}_{q-1}), \mathcal{P}_q). \tag{5.2}$$

5.2 Concurrent Execution

When executing two or more patterns concurrently, we actually have to consider the fact that they are competing for the same cache. The number of total cache misses will be higher than just the sum of the individual cache miss rates. The reason for this is, that the patterns will mutually evict cache lines from the cache due to alignment conflicts. To which extend such conflict misses occur does not only depend on the patterns themselves, but also on the data placement and details of the cache alignment. Unfortunately, these parameters are not know during cost evaluation.

Hence, we model the impact of the cache interference between concurrent patterns by dividing the cache among all patterns. Each individual pattern gets only a fraction of the cache according to its *footprint size*. We define a pattern's footprint size $\mathbf{F}$ as the number of cache lines that it potentially revisits.

With single sequential traversals, a cache line is never visited again once access has moved on to the next cache line. Hence, simple sequential patterns virtually occupy only one cache line a at time. Or in other words, the number of cache misses is independent of the available cache size. The same holds for single random traversals with $R.w - u \geq B$. In all other cases, basic access patterns (potentially) revisit all cache lines covered by their respective data region. We define $\mathbf{F}$ as follows.

Be $\mathcal{P} = \mathsf{T}([..,]R, u) \in \mathbb{P}_\mathrm{b}$ a basic access pattern, then

$$\mathbf{F}_i(\mathcal{P}) = \begin{cases} 1, & \text{if} \quad \mathsf{T} = \mathsf{s\_trav}^x \\ 1, & \text{if} \quad \mathsf{T} = \mathsf{r\_trav} \ \wedge \ R.w - u \geq B_i \\ |R|_{B_i}, & \text{else.} \end{cases}$$

Be $\mathcal{P}_1, \ldots, \mathcal{P}_p \in \mathbb{P}$ ($p > 1$) access patterns, then

$$\begin{aligned} \mathbf{F}_i(\oplus(\mathcal{P}_1, \ldots, \mathcal{P}_p)) &= \max\{\mathbf{F}_i(\mathcal{P}_1), \ldots, \mathbf{F}_i(\mathcal{P}_p)\}, \\ \mathbf{F}_i(\odot(\mathcal{P}_1, \ldots, \mathcal{P}_p)) &= \sum_{q=1}^{p} \mathbf{F}_i(\mathcal{P}_q). \end{aligned}$$

Further, we use $\vec{\mathbf{M}}_{i/\nu}$ with $\nu \geq 1$ to denote the number of misses with only $\frac{1}{\nu}$th of the total cache size available. To calculate $\vec{\mathbf{M}}_{i/\nu}$, we simply replace C and $\#$ by $\frac{C}{\nu}$ and $\frac{\#}{\nu}$, respectively, in the formulas in Sections 4 and 5.1. Likewise, we define $\mathbf{S}_{i/\nu}(\mathcal{P})$. We write $\vec{\mathbf{M}}_i = \vec{\mathbf{M}}_{i/1}$ and $\mathbf{S}_i = \mathbf{S}_{i/1}$.

With these tools at hand, we calculate the cache misses for concurrent execution of patterns $\mathcal{P}_1, \ldots, \mathcal{P}_p \in \mathbb{P}$ ($p > 1$) given an initial cache state $\mathbf{S}$ as

$$\vec{\mathbf{M}}_{i/\nu}(\mathbf{S}_i, \odot(\mathcal{P}_1, \ldots, \mathcal{P}_p)) = \sum_{q=1}^{p} \vec{\mathbf{M}}_{i/\nu_q}(\mathbf{S}_i, \mathcal{P}_q) \tag{5.3}$$

with

$$\nu_q = \frac{\mathbf{F}(\odot(\mathcal{P}_1, \ldots, \mathcal{P}_p))}{\mathbf{F}(\mathcal{P}_q)} \cdot \nu.$$

After executing $\odot(\mathcal{P}_1, \ldots, \mathcal{P}_p)$, the cache contains a fraction of each data region involved, proportional to its footprint size:

$$\mathbf{S}_i(\odot(\mathcal{P}_1, \ldots, \mathcal{P}_p)) = \bigcup_{q=1}^{p} \mathbf{S}_{i/\nu_q}(\mathcal{P}_q)$$

with ν_q as defined before.

6. Experimental Validation

To validate our cost model, we will compare the estimated costs with experimental results. We focus on characteristic operations, here. The data access pattern of each operation is a combination of several basic patterns. The operations are chosen so that each basic pattern occurs at least once. Extension to further operations and whole queries, however, is straight forward, as it just means applying the same techniques to combine access patterns and derive their cost functions.

6.1 Setup

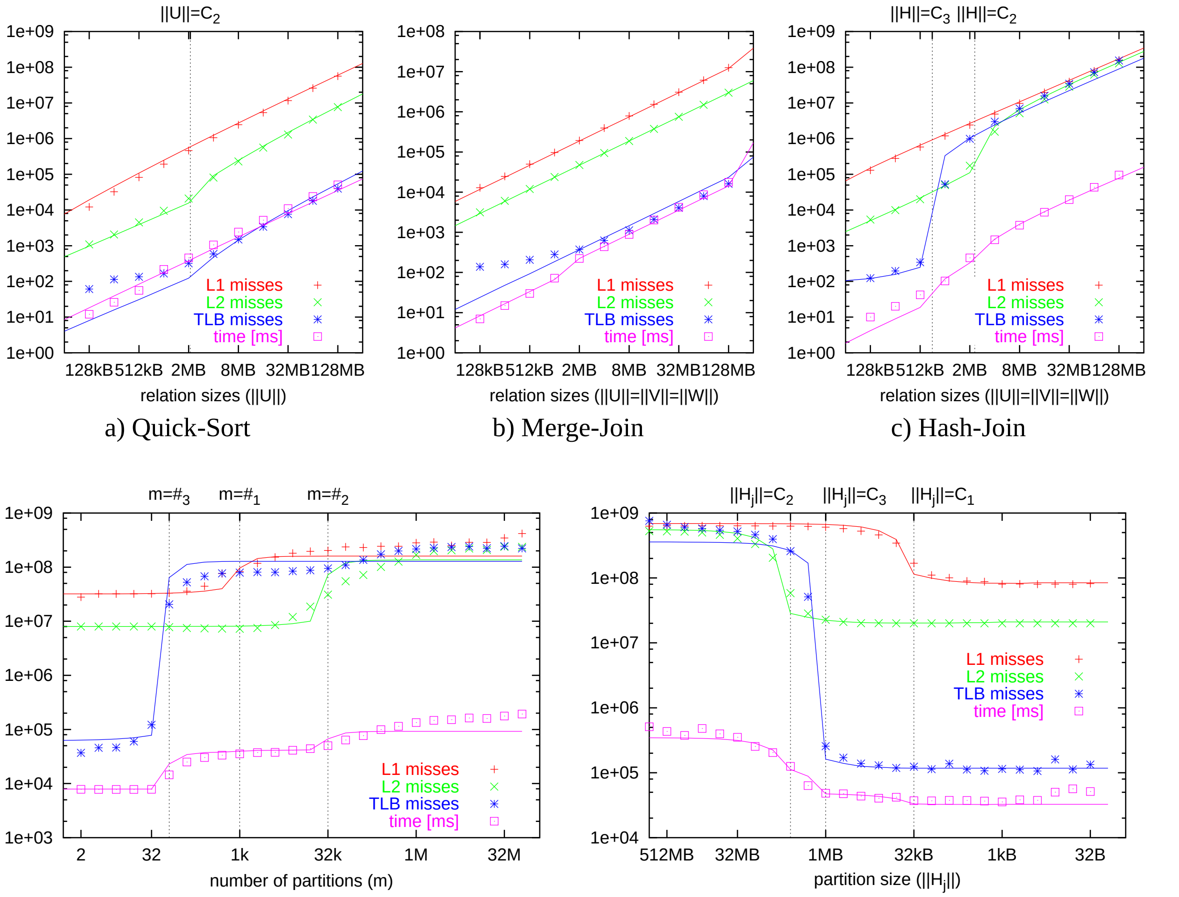
We implemented our cost functions and used our main-memory DBMS prototype Monet [BK99] as experimentation platform. We ran our experiments on an SGI Origin2000. Table 3 lists the relevant hardware features of the machine. The cache characteristics are measured with our calibration tool. We use the CPU's hardware counters to get the exact number of cache and TLB misses while running our experiments. Thus, we can validate the estimated cache miss rates. Validating the resulting total memory access cost (i.e., miss rates scored by their latencies) is more complicated, as there is no way to measure the time spent on memory access. We can only measure the total elapsed time, and this includes the (pure) CPU costs as well. Hence, we extend our model to estimate the total execution time T as sum of memory access time and pure CPU time

$$T = T_{\text{Mem}} + T_{\text{CPU}} \tag{6.1}$$

with T_{Mem} as in Equation (3.1). We calibrate T_{CPU} for each algorithm in an in-cache setting, i.e., without memory cost.

description		value
machine type		SGI Origin2000
OS		IRIX64 6.5
CPU		MIPS R10000
CPU speed		250 MHz
main-memory size		48 GB (4 GB local)
cache & TLB levels	N	3
L1 cache capacity	C_1	32 KB
L1 cache line size	B_1	32 bytes
L1 cache lines	$\#_1$	1024
L2 cache capacity	C_2	4 MB
L2 cache line size	B_2	128 bytes
L2 lines	$\#_2$	32,768
TLB entries	$\#_3$	64
page size	B_3	16 KB
TLB capacity $(\#_3 \cdot B_3)$	C_3	1 MB
TLB miss latency	$l_3^{\mathtt{r}} = l_3^{\mathtt{s}}$	228 ns = 57 cycles
sequential access		
L1 miss latency	$l_1^{\mathtt{s}}$	8 ns = 2 cycles
L2 miss latency	$l_2^{\mathtt{s}}$	188 ns = 47 cycles
L1 miss bandwidth	$b_1^{\mathtt{s}}$	3815 MB/s
L2 miss bandwidth	$b_2^{\mathtt{s}}$	555 MB/s
random access		
L1 miss latency	$l_1^{\mathtt{r}}$	24 ns = 6 cycles
L2 miss latency	$l_2^{\mathtt{r}}$	400 ns = 100 cycles
L1 miss bandwidth	$b_1^{\mathtt{r}}$	1272 MB/s
L2 miss bandwidth	$b_2^{\mathtt{r}}$	246 MB/s

Table 3: Hardware Characteristics



a) Quick-Sort b) Merge-Join c) Hash-Join

d) Partitioning ($||U|| = ||W|| = 512$ MB) e) Partitioned Hash-Join ($||U|| = ||V|| = ||W|| = 512$ MB)

Figure 7: Measured (points) and Predicted (lines) Cache Misses and Execution Time

6.2 Results

Figure 7 gathers our experimental results. Each plot represents one algorithm. The cache misses and times measured during execution are depicted as points. The respective cost estimations are plotted as lines. Cache misses are depicted in absolute numbers. Times are depicted in milliseconds. We will now discuss each algorithm in detail.

Quick-Sort Our first experiment is sorting. We use quick-sort to sort a table in-place. Quick-sort uses two cursors, one starting at the front and the other starting at the end. Both cursors sequentially walk towards each other swapping data items where necessary, until they meet in the middle. We model this as two concurrent sequential traversals, each sweeping over one half of the table: $\mathsf{s\_trav}^{\mathsf{s}}(U/2) \odot \mathsf{s\_trav}^{\mathsf{s}}(U/2)$ At the meeting point, the table is split in two parts and quick-sort recursively proceeds depth-first on each part. With n being the table's cardinality, the depth of the recursion is $\log_2 n$. In total, we model the data access pattern of quick-sort as

$$U \leftarrow quick\_sort(U): \qquad \oplus|_{i=1}^{log_2 U.n} \left(\oplus|_{j=i}^{log_2 U.n} \left(\mathsf{s\_trav}^{\mathsf{s}}(U/2^j) \odot \mathsf{s\_trav}^{\mathsf{s}}(U/2^j) \right) \right).$$

We varied the table sizes from 128 KB to 128 MB and the tables contained randomly distributed (numerical) data. Figure 7a shows that the models accurately predict the actual behavior. Only the start-up overhead of about 100 TLB misses is not covered, but this is negligible. The step in the L2 misses-curve depicts the effect of caching on repeated sequential access: Tables that fit into the cache have to be loaded only once during the top-level iteration of quick-sort. Subsequent iterations operate on the cached data, causing no additional cache misses.

Merge-Join Our next candidate is merge-join. Assuming both operands are already sorted, merge-join simply performs three concurrent sequential patterns, one on each input and one on the output:

$$W \leftarrow \mathit{merge\_join}(U,V): \qquad \mathsf{s\_trav}^{\mathsf{s}}(U) \odot \mathsf{s\_trav}^{\mathsf{s}}(V) \odot \mathsf{s\_trav}^{\mathsf{s}}(W).$$

Again, we use randomly distributed data and table sizes as before. In all experiments, both operands are of equal size, and the join is a 1:1-match. The respective results in Figure 7b demonstrate the accuracy of our cost functions. Further, we see that single sequential access is not affected by cache sizes. The costs are proportional to the data sizes.

Hash-Join While the previous operations perform only sequential patterns, we now turn our attention to hash-join. Hash-join performs random access to the hash-table, both while building it and while probing the other input against it. We model the data access pattern of hash-join as

$$W \leftarrow \mathit{hash\_join}(U,V): \qquad \mathsf{s\_trav}^{\mathsf{s}}(V) \odot \mathsf{r\_trav}(H) \oplus \mathsf{s\_trav}^{\mathsf{s}}(U) \odot \mathsf{r\_acc}(U.n, H) \odot \mathsf{s\_trav}^{\mathsf{s}}(W).$$

The plots in Figure 7c clearly show the significant increase in L2 and TLB misses, once the hash-table size $||H||$ exceeds the respective cache size.[7] Our cost model correctly predicts these effects and the resulting execution time.

Partitioning One way to prevent the performance decrease of hash-join on large tables is to partition both operands on the join attribute and then hash-join the matching partitions [SKN94, MBK00a]. If each partition fits into the cache, no additional cache misses will occur during hash-join.

Partitioning algorithms typically maintain a separate output buffer for each result partition. The input is read sequentially, and each tuple is written to its output partition. Data access within each output partition is also sequential. Hence, we model partitioning using a sequential traversal for the input and an interleaved multi-cursor access for the output:

$$\{U_j\}|_{j=1}^{m} \leftarrow \mathit{cluster}(U,m): \qquad \mathsf{s\_trav}^{\mathsf{s}}(U) \odot \mathsf{nest}(\{U_j\}|_{j=1}^{m}, m, \mathsf{s\_trav}^{\mathsf{s}}(U_j), \mathtt{ran})$$

The curves in Figure 7d demonstrate the effect we discussed in Section 4.7: The number of cache misses increases significantly, once the number of output buffers m exceeds the number of available cache blocks $\#$. Though they tend to under estimate the costs for very high numbers of partitions, our models accurately predict the crucial points.

Partitioned Hash-Join Once the inputs are partitioned, we can join them by performing a hash-join on each pair of matching partitions. We model the data access pattern of partitioned hash-join as

$$\{W_j\}|_{j=1}^{m} \leftarrow \mathit{part\_hash\_join}(\{U_j\}|_{j=1}^{m}, \{V_j\}|_{j=1}^{m}, m): \qquad \oplus|_{j=1}^{m}(\mathsf{hash\_join}(V_j, U_j, W_j))$$

Figure 7e shows that the cache miss rates and thus the total cost decrease significantly, once each partition (respectively its hash-table) fits into the cache.

[7] The plots show no such effect for L1 misses, as all hash-tables are larger than the L1 cache, here.

7. Conclusion

We presented a new generic approach to build generic database cost models for hierarchical memory systems.

We extended the knowledge base on analytical cost-models for query optimization with a strategy derived from our experimentation with main-memory database technology. The approach taken shows that we can achieve hardware-independence by modeling hierarchical memory systems as multiple level of caches. Each level is characterized by a few parameters describing its sizes and timings. This abstract hardware model is not restricted to main-memory caches. As we pointed out, the characteristics of main-memory access characteristics are very similar to those of disk access. Viewing main-memory (e.g., a database system's buffer pool) as cache for disk access, it is obvious that our approach also covers I/O. As such, the model presented provides a valuable addition to the core of cost-models for disk-resident databases as well.

Adaptation of the model to a specific hardware is done by instantiating the parameters with the respective values of the very hardware. Our *Calibrator*, a software tool to measure these values on arbitrary systems, is available for download from our web site `http://monetdb.cwi.nl`.

We identified a few key access patterns eminent in the majority of relational algebra implementations. The key patterns fulfill two major requirements: they are simple and they have a relevant impact on data access costs. For these basic patterns, we developed cost functions that estimate the respective access cost in terms of cache misses scored by there latency. To maintain hardware-independence, the functions are parameterized with the hardware characteristics.

We introduced two operators to combine simple patterns to more complex patterns and developed rules how to generate the respective cost functions.

With our approach, building physical costs function for database operations boils down to describing the algorithms' data access in a kind of ”pattern language” as presented in Section 3.3. This task requires only information that can be derived from the algorithm. Especially, no knowledge about the hardware is needed, here. The detailed cost function are than automatically derived from the pattern descriptions.

Though focusing on data access costs, here, our model does not ignore CPU costs. For now, the pure CPU costs of each algorithm are once measured ”by hand”. Proper techniques to automate this process are subject to future research, making-up one of the top positions in our agenda.

References

[ADHS01] A. G. Ailamaki, D. J. DeWitt, M. D. Hill, and M. Skounakis. Weaving Relations for Cache Performance. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, pages 169–180, Rome, Italy, September 2001.

[ADHW99] A. G. Ailamaki, D. J. DeWitt, M. D. Hill, and D. A. Wood. DBMSs on a Modern Processor: Where does time go? In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, pages 266–277, Edinburgh, Scotland, UK, September 1999.

[BK99] P. A. Boncz and M. L. Kersten. MIL Primitives for Querying a Fragmented World. *The VLDB Journal*, 8(2):101–119, October 1999.

[BMK99] P. A. Boncz, S. Manegold, and M. L. Kersten. Database Architecture Optimized for the New Bottleneck: Memory Access. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, pages 54–65, Edinburgh, United Kingdom, September 1999.

[Gra94] G. Graefe. *Query Processing Techniques for Large Databases.* to be published, 1994.

[HS89] M. D. Hill and A. J. Smith. Evaluating Associativity in CPU Caches. *IEEE Transactions on Computers (TOC)*, 38(12), December 1989.

[IP99] Y. E. Ioannidis and V. Poosala. Histogram-Based Approximation of Set-Valued Query-Answers. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, pages 174–185, Edinburgh, Scotland, UK, September 1999. Morgan Kaufmann.

[LN96] S. Listgarten and M.-A. Neimat. Modelling Costs for a MM-DBMS. In *Proceedings of the International Workshop on Real-Time Databases, Issues and Applications (RTDB)*, pages 72–78, Newport Beach, CA, USA, March 1996.

[MBK00a] S. Manegold, P. A. Boncz, and M. L. Kersten. Optimizing Database Architecture for the New Bottleneck: Memory Access. *The VLDB Journal*, 9(3):231–246, December 2000.

[MBK00b] S. Manegold, P. A. Boncz, and M. L. Kersten. What happens during a Join? — Dissecting CPU and Memory Optimization Effects. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, pages 339–350, Cairo, Egypt, September 2000.

[MBK02] S. Manegold, P. A. Boncz, and M. L. Kersten. Optimizing Main-Memory Join On Modern Hardware. *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, ?(?), ? 2002. To appear.

[PIHS96] V. Poosala, Y. E. Ioannidis, P. J. Haas, and E. J. Shekita. Improved Histograms for Selectivity Estimation of Range Predicates. In *Proceedings of the ACM SIGMOD International Conference* *on Management of Data (SIGMOD)*, pages 294–305, Montreal, Quebec, Canada, June 1996. ACM Press.

[RR98] J. Rao and K. A. Ross. Reusing Invariants: A New Strategy for Correlated Queries. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD)*, pages 37–48, Seattle, WA, USA, June 1998. ACM Press.

[RR99] J. Rao and K. A. Ross. Cache Conscious Indexing for Decision-Support in Main Memory. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, pages 78–89, Seattle, WA, USA, September 1999. Morgan Kaufmann.

[RR00] J. Rao and K. A. Ross. Making B^+-Trees Cache Conscious in Main Memory. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD)*, pages 475–486, Dallas, TX, USA, May 2000. ACM Press.

[SKN94] A. Shatdal, C. Kant, and J. Naughton. Cache Conscious Algorithms for Relational Query Processing. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, pages 510–512, Santiago, Chile, September 1994.

[WK90] K.-Y. Whang and R. Krishnamurthy. Query Optimization in a Memory-Resident Domain Relational Calculus Database System. *ACM Transactions on Database Systems (TODS)*, 15(1):67–95, March 1990.