Computer Science and Artificial Intelligence Laboratory
Technical Report

MIT-CSAIL-TR-2006-078 November 27, 2006

Materialization Strategies in a Column-Oriented DBMS

Daniel J. Abadi, Daniel S. Myers, David J. DeWitt, and Samuel R. Madden

massachusetts institute of technology, cambridge, ma 02139 usa — www.csail.mit.edu

Materialization Strategies in a Column-Oriented DBMS

Daniel J. Abadi
MIT
dna@csail.mit.edu

Daniel S. Myers
MIT
dsmyers@csail.mit.edu

David J. DeWitt
University of Wisconsin
dewitt@cs.wisc.edu

Samuel R. Madden
MIT
madden@csail.mit.edu

Abstract

There has been a renewed interest in column-oriented database architectures in recent years. For read-mostly query workloads like those found in data warehouse and decision support applications, "column-stores" have been shown to perform particularly well relative to "row-stores". In order for column-stores to be readily adopted as a replacement for row-stores, they must maintain the same interface to external applications as do row-stores. This implies that column-stores must output row-store style tuples.

Thus, the input columns stored on disk must be converted to rows at some point in the query plan. The optimal point at which to do this is not obvious. This problem can be considered as the opposite of the projection problem in row-store systems. While rows-stores need to determine where in query plans to place projection operators to make tuples narrower, column-stores need to determine when to combine single-column projections into wider tuples. This paper describes a variety of strategies for tuple construction and intermediate result representations, and then provides a systematic evaluation of these strategies.

1 Introduction

Vertical partitioning has long been recognized as a valuable tool for increasing the performance of read-intensive databases. Recent years have seen the emergence of several database systems that take this idea to the extreme by fully vertically partitioning database tables and storing them as columns on disk [16, 7, 11, 1, 10, 14, 13]. Research on these *column-stores* has shown that for certain read-mostly workloads, this approach can provide substantial performance benefits over traditional row-oriented database systems. Most column-stores choose to offer a standards-compliant relational database interface (e.g., ODBC, JDBC, etc.) This means they must ultimately stitch together separate columns into tuples of data that are output. Determining the tuple construction point in a query plan is the inverse of the problem of applying projections in a row-oriented database, since rather than deciding when to project an attribute out of an intermediate result flowing through the query plan, the system must decide when to add it in. Lessons from when to apply projection in row-oriented databases (projections are almost always performed as soon as an attribute is no longer needed) suggest a natural tuple construction policy: at each point where a column is accessed, add the column to an intermediate tuple representation if that column is needed by some later operator or is included in the set of output columns. Then, at the top of the query plan, these intermediate tuples can be directly output to the user. We call this process of adding columns to intermediate results *materialization*, and call the simple scheme described above *early materialization*, since it seeks to form intermediate tuples as early as possible.

Surprisingly, we have found that early materialization is not always the best strategy to employ in a column store. To illustrate why this is the case, consider a simple example: suppose a query consists of three selection operators σ_1, σ_2, and σ_3 over three columns, $R.a$, $R.b$, and $R.c$ (all sorted in the same order and stored in separate files), where σ_1 is the most selective predicate and σ_3 is the least selective. An early materialization strategy could process this query as follows: Read in a block of $R.a$, a block of $R.b$, and a block of $R.c$ from disk. Stitch them together into (likely more than one) block of row-store style triples $(R.a, R.b, R.c)$. Apply σ_1, σ_2, and σ_3 in turn, allowing tuples that match the predicate to pass through.

However, there is another strategy that can be more efficient; we call this second approach *late materialization*, because it doesn't form tuples until after some part of the plan has been processed. It works as follows: first scan $R.a$ and output the positions (ordinal offsets of values within the column) in $R.a$ that satisfy the σ_1 (these positions can take the form of ranges, lists, or a bitmap). Repeat with $R.b$ and $R.c$, outputting positions that satisfy σ_2 and σ_3 respectively.

Next, use position-wise AND operations to intersect the position lists. Finally, re-access $R.a$, $R.b$, and $R.c$ and extract the values of the records that satisfied all predicates and stitch these values together into output tuples. This late materialization approach can potentially be more CPU efficient because it requires fewer intermediate tuples to be stitched together (which is a relatively expensive operation as it can be thought of as a join on position) and position lists are a small, very compressible data structure that can be operated on directly with very little overhead. For example, 32 (or 64 depending on processor word size) positions can be intersected at once when ANDing together two position lists represented as bit-strings. Note, however, that one problem of this late materialization approach is that it requires re-scanning the base columns to form tuples, which can be slow (though they are likely to still be in memory upon re-access if the query is properly pipelined).

In this paper, we study the use of early and late materialization in the C-Store DBMS. We focus on standard warehouse-style queries: read-only workloads, with selections, aggregations, and joins. We study how different selectivities, compression techniques, and query plans affect the trade-offs in alternative materialization strategies. We run experiments to determine when one approach dominates the other, and develop an analytical model that can be used, for example, in a query optimizer to select a materialization strategy. Our results show that, on some workloads, late materialization can be an order of magnitude faster than early-materialization, while on other workloads, early-materialization outperforms late-materialization by an order of magnitude.

In the remainder of this paper we give a brief overview of the C-Store query executer in Section 1.1. We then detail the trade-offs between the two fundamental materialization strategies in Section 2, and then present both pseudocode and an analytical model for some query plans using each strategy in Section 3. We then validate our models with experimental results by extending C-Store to support query execution using any of the materialization strategies proposed in Section 4. We then describe related work in Section 5 and conclude in Section 6.

1.1 The C-Store Query Executor

We now provide a brief overview of the C-Store query executor, which is more fully described in [16, 3] and available in an open source release [2]. The components of the query executor relevant to the present study are the on-disk layout of data, the access methods provided for reading data from disk, the data structures provided for representing data in the DBMS, and the operators provided for manipulating these data.

Each column is stored in a separate file on disk as a series of 64KB blocks and can be optionally encoded using a variety of compression techniques. In this paper we experiment with column-specific compression techniques (run-length encoding and bit-vector encoding), and with uncompressed columns. In a run-length encoded file, each block contains a series of RLE triples (V, S, L), where V is the value, S is the start position of the run, and L is the length of the run.

A bit-vector encoded file representing a column of size n with k distinct values consists of k bit-strings of length n, one per unique value, stored sequentially. Bit-string k has a 1 in the i^{th} position if the column it represents has the value k in the i^{th} position.

C-Store provides an access method (or *DataSource*) for each encoding type. All C-Store data sources support two basic operations: reading positions from a column and reading $(position, value)$ pairs from a column (note that any given instance of a data source can be used to read positions or $(position, value)$ pairs, but not both). Additionally, all C-Store data sources accept SARGable predicates [15] to restrict the set of results returned. In order to minimize CPU overhead, C-Store data sources and operators are block-oriented. Data sources return data from the underlying files in blocks of encoded data, wrapped inside a C++ class that provides iterator-style (hasNext() and getNext() methods) and vector-style [7] (asArray()) access to the data in the blocks.

In the Section 3.1 we will give pseudocode for three basic C-Store operators: DataSource (Select), AND, and Merge. We also describe the Join operator in section 4.3). The DataSource operator reads in a column of data and produces the column values that pass a predicate. AND accepts input position lists and produces an output position list representing their intersection. Finally, the n-ary Merge operator combines n inputs of $(position, value)$ pairs into a single output of n-attribute tuples.

2 Materialization Strategy Trade-offs

In this section we present some of the trade-offs that are made between materialization strategies. A materialization strategy needs to be in place whenever more than one attribute from any given relation is accessed (which is the case for most queries). Since a column-oriented DBMS stores each attribute independently, it must have some mechanism for stitching together multiple attributes from the same logical tuple into a physical tuple. Every proposed column-oriented architecture accomplishes this by attaching either physical or virtual *tuple identifiers* or *positions* to column values. To reconstruct a tuple from multiple columns of a relation, the DBMS simply needs to find matching positions. Modern column-oriented systems [16, 6, 7] store columns in position order; i.e., to reconstruct the first tuple one needs to take the first value from each column, the second tuple is constructed from the second value from each column, and likewise as one iterates down through the columns. This accelerates the tuple reconstruction process.

As described in the introduction, tuple reconstruction can occur at different points in a query plan. *Early materialization* constructs tuples as soon as (or sometimes before) tuple values are needed in the query plan. *Late materialization* constructs tuples as late as possible, sometimes even at the query output. Each approach has a set of advantages.

2.1 Late Materialization Advantages

Late materialization allows the executer to operate on positions and compressed, column-oriented data and defer tuple construction.

2.1.1 Operating on Positions

The process of tuple construction starts to get interesting as soon as predicates are applied to different columns. The result of predicate application are different subsets of positions for different columns. Tuple reconstruction thus requires a equi-join on position of multiple columns. However, since columns are sorted by position, this join can be performed with a relatively fast merge join.

Positions can be represented using a variety of compression techniques. *Runs* of consecutive positions can be represented using position ranges of the form [startpos, endpos]. Positions can also be represented as bitmaps using a single bit to represent every position in a column, with a '1' in the bit-map entry if the tuple at the position passed the predicate and a '0' otherwise. For example, for a position range of 11-20, a bit-vector of 0111010001 would indicate that positions 12, 13, 14, 16, and 20 contained values that passed the predicate.

It turns out that, in many cases, these position representations can be operated on directly without using column values. For example, an AND operation of 3 single column predicates in the WHERE clause of an SQL query can be performed by applying each predicate separately on its respective column to produce 3 sets of positions for which the predicate matched. These 3 position lists can be intersected to create a new position list that contains a list of all positions of tuples that passed every predicate. This position list can then be sent to other columns in the same relation to retrieve additional column values from those logical tuples, which can then be sent to parent operators in the query plan for processing.

Position operations are highly efficient from a CPU perspective due to the highly compressible nature of position representations and the ease of operation on them. For example, intersecting two position lists represented using bit-strings is requires only n/32 (or n/64 depending on processor word size) instructions (if n is the number of positions being intersected) since 32 positions can be intersected in a single instruction. Intersecting a position range with a bit-string is even faster (requiring a constant number of instructions), as the result is equal to the subset of the same bit-string starting at the beginning of the position range and ending at the last position covered by the range.

Not only is the processing of positions fast, but their creation can also be fast since in many cases the positions of tuples that pass a predicate can be derived directly from a column index. For example, if there is a clustered index over a column and a predicate on a value range, the index can be accessed to find the start and end positions that match the value range, and these two positions can encode the entire set of positions in that column that match the predicate. Similarly, there might be a bit-map index on that column [12, 16, 3] in which case the positions matching a predicate can be derived by ORing together the appropriate bitmaps. In both cases, the original column values never have to be accessed.

2.1.2 Column-Oriented Data Structures

Another advantage of late materialization is that column values can be stored together contiguously in memory in column-oriented data structures. This has two performance advantages: First, the column can be kept compressed in memory using the same column-oriented compression techniques as was used to store the column on disk. [3] showed that techniques such as run length encoding (RLE) of column values and bit-vector encoding are ideally suited for column stores and can easily be operated on directly. For example, for RLE encoded data, an entire run length of values can be processed in one operator loop. Tuple construction requires decompression of run-length encoded data since only the values in one column tend to repeat, not the entire tuple (i.e., a table with 5 tuples: (2, a), (2, b), (2, c), (2, d), (2, e) can be represented in column format as (2,5), (a,b,c,d,e) where the (2,5) indicates that the value 2 repeats 5 times; however tuple construction requires the value '2' to appear in each of the five tuples in which it is contained).

Second, looping through values from a column oriented data structure tends to be much faster than looping through values using a tuple iterator interface. This is attributed to two main reasons: First, entire cache lines are filled with values from the same column. This maximizes the efficiency of the memory bandwidth bottleneck [4] as the cache prefetcher only fetches relevant data. Second, high IPC (instructions-per-cycle) vector processing code can be written for column block access taking advantage of modern super-scalar CPUs [6, 5, 7].

2.1.3 Construct Only Relevant Tuples

In many cases, a query outputs fewer tuples than are actually processed. Predicates usually reduce the number of tuples output, and aggregations combine tuples together into summary tuples. Thus, if the executer waits long enough before constructing a tuple, it might be able to avoid constructing it altogether.

2.2 Early Materialization Advantages

The fundamental problem with waiting as long as possible before tuple construction is that in some cases columns have to be accessed more than once in a query plan. Take, for example, the case where a column is accessed once to get positions of the column that match a predicate and again downstream in the query plan for its values. For the cases where the positions that match the predicate can not be answered directly from an index, the column values must be accessed twice. If the query is properly pipelined, the re-access will not have a disk cost component (the disk block will still be in the buffer cache); however there will be a CPU cost component of scanning through the block to find the set of values corresponding to a given set of positions. This cost will be even higher if the positions are not in sorted order (if, for example, they got reordered through a join - an example of this is given in Section 4.3).

For the early materialization strategy, as soon as a column is accessed, it's values are added to the tuple being constructed and the column will not need to be reaccessed. Thus, the fundamental trade-off between early materialization and late materialization is the following: while late materialization enables several performance optimizations (operating directly on position data, only constructing relevant tuples, operating directly on column-oriented compressed data, and high value iteration speeds), if the column re-access cost at tuple reconstruction time is high, a performance penalty is paid.

3 Query Processor Design

Having described some of the fundamental trade-offs between early and late materialization, we now give detailed examples for how these materialization alternatives translate into query execution plans in a column-oriented system. We present query plans for simple selection queries in C-Store and give both pseudocode and an analytical model for each materialization strategy.

3.1 Operator Implementation and Analysis

To better illustrate the trade-offs between early and late materialization, in this section we present an analytical model of the two strategies. The model is composed of three basic types of operators:

- Data source (DS) operators that read columns from disk, filtering on one or more single-column predicates or a position list as they go, and producing vectors of positions or tuples of positions and values.

$\lvert C_i \rvert$	No. of disk blocks in Col_i
$\lvert\lvert C_i \rvert\rvert$	No. of "tuples" in Col_i
$\lvert\lvert POSLIST \rvert\rvert$	No. of positions in $POSLIST$
F	Fraction of pages of a column in buffer pool
SF	Selectivity factor of predicate
BIC	CPU time in ms of getNext() call in block iterator
TIC_{TUP}	CPU time for getNext() call in tuple iterator
TIC_{COL}	CPU time for getNext() call in column iterator
FC	Time for a function call
PF	Prefetch size (in number of disk blocks)
$SEEK$	Disk seek time
$READ$	Time to read a block from disks
RL	Average length of a sorted run in RLE encoded columns (RL_c) or position lists (RL_p) (equal to one if uncompressed)

Table 1: Notation used in analytical model

- AND operators that merge several filtered position vectors or tuples with positions together into a smaller list of positions.
- Tuple construction operators that combine narrow tuples of positions and values into wider columns with positions and multiple values.

We use the notation in Table 1 to describe the costs of the different operators.

3.2 Data Sources

In this section, we consider the cost of accessing a column on disk via a data source operator. We consider four cases:

Case 1: A column C_i of $|C_i|$ blocks is read from disk and a predicate with selectivity SF is applied to each tuple. The output is a column of positions. The pseudocode and cost analysis of this case is shown in Figure 1.

```
DS_Scan-Case1(Column C, Pred p)
 1. for each block b in C
 2.   read b from disk
 3.   for each RLE tuple t in b
 4.     apply p to t
 5.     output positions from t
```

$$CPU =$$
$$|C_i| * BIC+ \qquad (1)$$
$$||C_i|| * (TIC_{COL} + FC)/RL+ \qquad (3,4)$$
$$SF * ||C_i|| * FC \qquad (5)$$
$$IO = (\frac{|C_i|}{PF} * SEEK + |C_i| * READ) * (1 - F) \qquad (2)$$

Figure 1: Psuedocode and cost formulas for data sources, Case 1. Numbers in parentheses in cost formula indicate corresponding steps in the pseudocode.

Case 2: A column C_i of $|C_i|$ blocks is read from disk and a predicate with selectivity SF is applied to each tuple. The output is a column of (positions, value) pairs.

The cost of Case 2 is identical to Case 1 except for step (5) which becomes $SF * ||C_i|| * (TIC_{TUP} + FC)$. The slightly higher cost reflects the cost of gluing positions and values together for the output.

Case 3: A column C_i of $|C_i|$ blocks is read from disk or memory and filtered with a list of positions, $POSLIST$. The output is a column of the values corresponding to those positions. The pseudocode and cost analysis of this case is shown in Figure 2.

```
DS_Scan-Case3(Column C, POSLIST pl)
 1. for each block b in C
 2.  read b from disk
 3.  iterate through pl, for each pos. (range)
 4.   jump to pos (range) in b and output value(s)
```

$CPU =$

$$|C_i| * BIC+ \tag{1}$$
$$||POSLIST||/RL_p * (TIC_{COL})+ \tag{3}$$
$$||POSLIST||/RL_p * (TIC_{COL} + FC) \tag{4}$$
$$IO = (\frac{|C_i|}{PF} * SEEK + SF * |C_i| * READ) * (1 - F) \tag{2}$$

/* F=1 and IO $\rightarrow$ 0 if col already accessed */

/* $SF * |C_i|$ is a lower bound for the number of necessary blocks to read in. However, for highly localized data (like the semi-sorted data we will work with), this is a reasonable approximation*/

Figure 2: Psuedocode and cost formulas for data sources, Case 3.

Case 4: A column C_i of $|C_i|$ blocks is read from disk and a set of tuples EM_i of the form $(pos, < a_1, \ldots, a_n >)$ is input to the operator. The operator jumps to the position *pos* in the column and a predicate with selectivity SF is applied. Tuples that satisfy the predicate are merged with EM_i to create a tuple of the form $(pos, < a_1, \ldots, a_n, a_{n+1} >)$ that contains only the positions that were in EM_i and that satisfied the predicate over C_i. The pseudocode and cost analysis of this case is shown in Figure 3.

```
DS_Scan-Case4(Column C, Pred p, Table EM)
 1. for each block b in C
 2. read b from disk
 3. iterate through tuples e in EM, extract pos
 4. use pos to jump to correct tuple t in C
    and apply predicate
 5. if predicate succeeded, output <e, t>
```

$CPU =$

$$|C_i| * BIC+ \tag{1}$$
$$||EM_i|| * TIC_{TUP}+ \tag{3}$$
$$||EM_i|| * ((FC + TIC_{TUP}) + FC) \tag{4}$$
$$SF * ||EM_i|| * (TIC_{TUP}) \tag{5}$$
$$IO = (\frac{|C_i|}{PF} * SEEK + |C_i| * READ) * (1 - F) \tag{2}$$

Figure 3: Psuedocode and cost formulas for data sources, Case 4.

3.3 Multicolumn AND

The AND operator can be thought of as taking in k position lists, $inpos_1 \ldots inpos_k$ and producing a new list of positions that represents the intersection of the two input lists, *outpos*. This model examines two possible representations for a list of positions: a list of ranges of positions(e.g., 1-100, 200-250) and bit-strings with one bit per position. The AND operator is only used in the LM case, since EM always uses Data scan Case 4 above to construct tuples as they are read from disk. If the positional input to AND are all ranges, then it will output position ranges. Otherwise it will output positions in bit-string format.

We consider three cases:

Case 1: Range inputs, Range output

In this case, each of the input position lists and the output position list are each encoded as ranges. The pseudocode and cost analysis for this case is shown in Figure 4. Since this operator is a streaming operator it incurs no I/O.

```
AND(POSLIST inpos 1 ... inpos k)
 1. iterate through position lists
 2. perform AND operations
 3. produce output lists
```

Let $M = max(||inpos_i||/RL_{p_i}, i \in 1 \ldots k)$

$COST =$

$$\sum_{i=1 \ldots k} (TIC_{COL} * ||inpos_i||/RL_{p_i})+ \tag{1}$$
$$M * (k - 1) * FC+ \tag{2}$$
$$M * TIC_{COL} * FC \tag{3}$$

Figure 4: Psuedocode and cost formulas for AND, Aase 1.

Case 2: Bit-list inputs, bit-list output

In this case each of the input position lists and the output position lists are bit vectors. The input position lists are "ANDed" 32 bits at a time. The cost formula for this case is identical to Case 1 except that every instance of $||inpos_i||/RL_{p_i}$ in Figure 4 is replaced with $||inpos_i||/32$ (32 is the processor word size).

Case 3: Mix of Bit-list and range inputs, bit-list output

In this case the input position lists to the AND operator are a mix of range and bit lists. The output is a bit-list. Execution occurs in three steps. First the range lists are intersected to produce one range list. Second, the bit-lists are anded together to produce a single bit list. Finally, the single range and bit lists are combined to produce the output list.

3.4 Tuple Construction Operators

The final two operators we consider are tuple construction operators. The first, the MERGE operator, takes k sets of values $VAL_1 \ldots VAL_k$ and produces a set of k-ary tuples. This operator is used to construct tuples at the top of an LM plan. The pseudocode and cost of this operation is shown in Figure 5.

```
Merge(Col s1,..., Col sk)
 1. iterate through all k cols of len. ||VAL_i||
 2. produce output tuples
```

$COST =$

// Access values as vector (don't use iterator)

$$||VAL_i|| * k * FC + \quad (1)$$

// Produce tuples as array (don't use iterator)

$$||VAL_i|| * k * FC) \quad (2)$$

Figure 5: Psuedocode and cost formulas for Merge.

This analysis assumes that we have an iterator over each of the input streams, that all of the input streams are memory resident, and that only one iterator is needed to produce the output stream.

The second tuple construction operator is the SPC (Scan, Predicate, and Construct) operator which can sit at the bottom of EM plans. SPC takes a set of columns $VAL_1 \ldots VAL_k$, reads them off disk, optionally takes a set of predicates to apply on the column values, and constructs tuples if all predicates pass. The pseudocode and cost of this operation is shown in Figure 6.

```
Merge(Col c1,..., Col ck, Pred p1,..., Pred pk)
 1. for each column, ||C_i||
 2.   for each block b in C_i
 3.     read b from disk
 4.     check predictates
 5.     construct tuples and output
```

$CPU =$

$$|C_i| * BIC + \quad (2)$$

$$||C_i|| * FC * \prod_{j=1\ldots(i-1)} (SF_j) + \quad (4)$$

$$||C_k|| * TIC_{TUP} * \prod_{j=1\ldots k} (SF_j) + \quad (5)$$

$$IO = (\frac{|C_i|}{PF} * SEEK + |C_i| * READ) \quad (3)$$

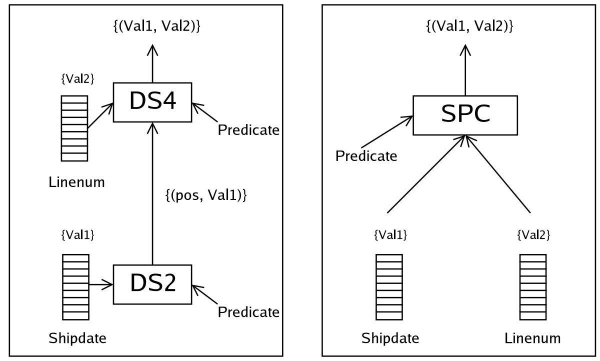
Figure 6: Psuedocode and cost formulas for SPC.

3.5 Example Query Plans

The use of these operators is illustrated in Figures 7 and 8 for the query:

```
SELECT shipdate, linenum
FROM   lineitem
WHERE  shipdate < CONST1
       AND linenum < CONST2
```

where lineitem is a C-Store projection consisting of the columns $return\_flag, shipdate, linenum$, and $quantity$. The primary and secondary sort keys for the lineitem projection are $returnflag$ and $shipdate$, respectively.



(a) (b)

Figure 7: Query plans for EM-piplined (a) and EM-parallel (b) strategies

One EM query plan, shown in Figure 7(a), uses a DS2 operator (Data Scan Case 2) operator to scan the shipnum column, producing a stream of $(pos, shipdate)$ tuples that satisfy the predicate $shipdate < CONST1$. This stream is used as one input to a DS4 operator along with the linenum column and the predicate $linenum < CONST2$ to produce a stream of $(shipnum, linenum)$ result tuples.

Another possible EM query plan, shown in Figure 7(b), contructs tuples at the very beginning of the plan - merging all needed columns at the leaf node as it applies the predicates using a SPC operator. The key difference between these early materialization strategies is that while the latter strategy has to scan and process all blocks for all input columns, the former plan applies each predicate in turn, and will construct a tuple incrementally, adding one attribute per operator. For non-selective predicates this is more work, but for selective predicates only subsets of blocks need to be processed (or in some cases the entire block can be skipped). We call the former strategy EM-pipelined and the latter strategy EM-parallel. The choice of which EM plan to use depends on the selectivity of the predicates (the former strategy likely is better if there are highly selective predicates).

Similarly to EM, there are both pipelined and parallel late materialization strategies. LM-pipelined is shown in Figure 8(a) and LM-parallel is shown in Figure 8(b). LM-parallel begins with two DS1 operators, one for the shipnum and linenum columns. Each DS1 operator scans its column, applying the appropriate predicate to produce a position list of those values that satisfy the predicate. The two position lists are streamed into an AND operator which intersects the two lists. The output position list is then streamed into two DS3 operators to obtain the corresponding values from the shipnum and linenum columns. As these values are obtained they are streamed into a merge operator to produce a stream of (shipnum, linenum) result tuples.

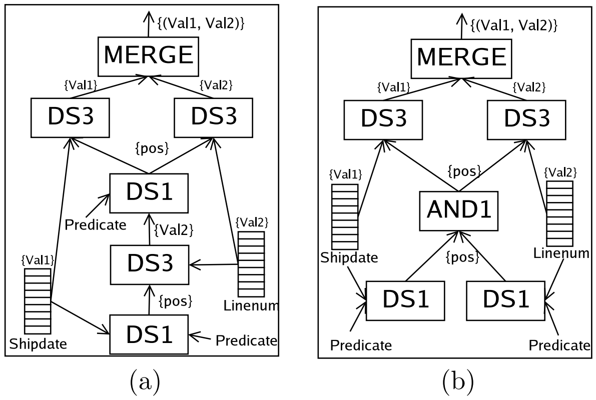


Figure 8: Query plans for LM-piplined (a) and LM-parallel (b) strategies

LM-pipelined works similarly to LM-parallel, except that it applies the DS1 operators one at a time, pipelining the positions of the shipdate values that passed the shipdate predicate to a DS3 operator for the linenum column which produces the column values at these input set of positions and sends these values to the linenum DS1 operator which only needs to apply its predicate to this value subset (rather than at all linenum positions). As a result, the need for the AND operator is obviated.

3.6 LM Optimization: Multi-Columns

Note that in DS Case 3, used in LM strategies to produce values from positions, the I/O cost is assumed to be zero if the column has already been accessed earlier in the plan, even if the column size is larger than available memory. This is made possible through a specialized data structure for representing intermediate results, designed to facilitate query pipelining, that allows blocks of column data to remain in user memory space after the first access so that it can be easily reaccessed again later on. We call this data structure a *multi-column*.

A multi-column contains a memory-resident, horizontal partition of some subset of attributes from a particular relation. It consists of:

A *covering position range* indicating the virtual start position and end position of the horizontal partition (for example, a position range could indicate that rows numbered 1000-2000 are covered in this multi-column).

An array of *mini-columns*. A mini-column is the set corresponding values for a specified position range of a particular attribute (MonetDB [7] calls this a *vector*, PAX [4] calls this a *mini-page*). Using the previous example, a mini-column for column X would contain 1001 values - the 1000th-2000th values in this column. The *degree* of a multi-column is the size of the mini-column array which is the number of included attributes. Each mini-column is kept compressed the same way as it was on disk.

A *position descriptor* indicating which positions in the position range remain valid. Positions are made invalid as predicates are applied on the multi-column. The position descriptor may take one of three forms:

- *Ranged positions*: All positions between a specified start and end position are valid.
- *Bit-mapped positions*: A bit-vector of size equal to the multi-column covering position range is given, with a '1' at a corresponding position if that position is valid. For example, for a position coverage of 11-20, a bit-vector of 0111010001 would indicate that positions 12, 13, 14, 16, and 20 are valid.
- *Listed positions* A list of valid positions inside the covering position range is given. This is particularly useful when few positions inside a multi-column are valid. Multi-columns may be *collapsed* to contain only values at valid positions inside the mini-columns, in which case the position descriptor must be transformed to *listed positions* format if positional information is still necessary for upstream operators.

When a page from a column is read from disk (say, for example, by a DS1 operator), a mini-column is created (which is essentially just a pointer to the page in the buffer pool) and the position descriptor indicates that every value begins as being valid. The DS1 operator then iterates through the column, applying the predicate to each value, and produces a new list of positions that are valid. The multi-column then replaces its position descriptor with the new position list (the mini-column remains untouched).

The AND operater then takes two multi-columns with overlapping covering position ranges, and creates a new multi-column where the covering position range and position descriptor is equal to the intersection of the position range and position descriptors of the input multi-columns, and the set of mini-columns is equal to the union of the input set of mini-columns. Thus, ANDing multi-columns is in essence the same operation as the AND of positions shown in Section 3.3; the only difference is that on top of performing the intersection of the position lists, ANDing multi-columns must also copy pointers to mini-columns to the output multi-column, but this can be thought of as a zero-cost operation.

If the AND operator produces multi-columns rather than just positions as an input to a DS3 operator, then the operator does not need to reaccess the column, but rather can work directly on one multi-column block at a time - iterating through the appropriate mini-column to produce only those values whose positions are valid according to the position descriptor. Single multi-columns blocks are worked on in each operator iteration, so that column-subsets can be pipelined up the query tree. With this optimization, DS3 I/O cost for a reaccessed column can be assumed to be zero.

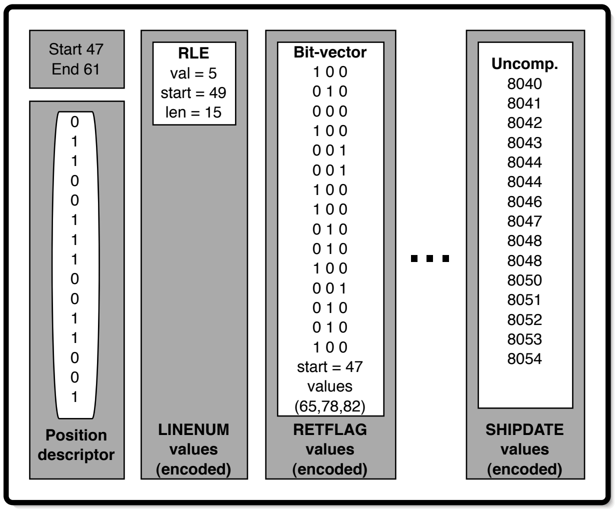


Figure 9: An example MultiColumn block containing values for the SHIPDATE, RETFLAG, and LINENUM columns. The block spans positions 47 to 61; within this range, positions 48, 49, 52, 53, 54, 57, 58, and 61 are active.

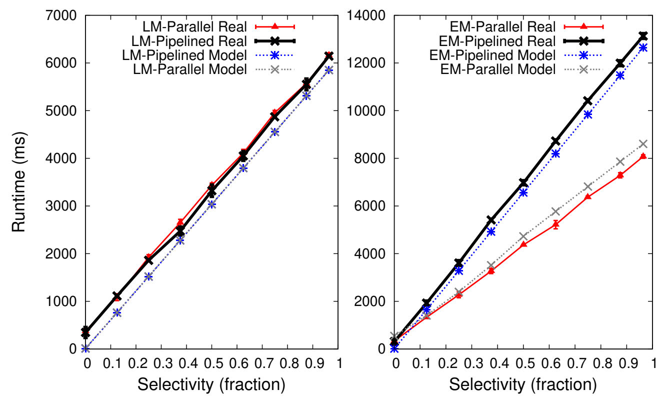
3.7 Predicted versus Actual Behavior

To gauge the accuracy of the analytical model we compared the predicated execution time for the selection query above with the actual execution time obtained using the C-Store prototype using a scale 10 version of the LineItem projection. The results obtained are presented in Figures 10(a) and 10(b), which plot response time as a function of the selectivity of the predicate $shipdate < CONST2$ for the the late and early materialization strategies respectively. The shipnum and linenum columns are both encoded using RLE encoding. The encoded sizes of the two columns are 1 block (3,800 tuples) and 5 blocks (26,726 tuples), respectively. Table 2 contains the constant values used by the analytical model which were obtained by running the small segments of code that only performed the variable in question (they were not reverse engineered to get the model to match the experiments). Both the model and the experiments incurred an additional cost at the end of the query to iterate through the output tuples ($numOutTuples * TIC_{TUP}$).

At this point, the important thing to observe is that the estimated performance from the EM and LM analytical models are quite accurate at predicting the actual performance of the C-Store prototype (at least for this query), providing a degree of reassurance regarding our understanding of how the two implementations actually work. A discussion on why the graphs look the way they do will be included in Section 4. We also modelled the same query but did not compress the linenum column (linenum was 60,000,000 tuples occupying 916 64KB blocks) and modelled different queries (including allowing both the shipdate and the linenum predicates to vary). We consistently found the model to reasonably predict our experimental results.

BIC	0.020 microsecs
TIC_{TUP}	0.065 microsecs
TIC_{COL}	0.014 microsecs
FC	0.009 microsecs
PF	1 block
$SEEK$	2500 microsecs
$READ$	1000 microsecs

Table 2: Constants used for Analytical Models



(a) (b)

Figure 10: Predicated and observed query performance for late (a) and early (b) materialization strategies on selection queries

4 Experiments

To evaluate the trade-offs between the early materialization and late materialization strategies, we ran two queries under a variety of configurations. These queries were run over data generated from the TPC-H dataset, a benchmark that models data typically found in decision support and data warehousing applications. Specifically, we generated an instance of the TPC-H data at scale 10, which yields a total database size of approximately 10GB with the biggest table (lineitem) containing 60,000,000 tuples. We then created a C-Store projection (which is a subset of columns from a table all sorted in the same order) of the SHIPDATE, LINENUM, QUANTITY, and RETURNFLAG columns; the projection was primarily sorted on RETURNFLAG, secondarily sorted on SHIPDATE, and tertiarily sorted on LINENUM. The RETURNFLAG and SHIPDATE columns were compressed using run-length encoding, the LINENUM column was stored redundantly using uncompressed, RLE, and bit-vector encodings, and the QUANTITY column was left uncompressed..

We ran the two queries on these data. First, we ran a simple selection query:

```
SELECT SHIPDATE, LINENUM
FROM LINEITEM
WHERE SHIPDATE < X AND
      LINENUM < Y
```

where X and Y are both constants. Second, we ran an aggregation version of this query:

```
SELECT SHIPDATE, SUM(LINENUM)
FROM LINEITEM
WHERE SHIPDATE < X AND
      LINENUM < Y
GROUP BY SHIPDATE
```

again with X and Y as constants. While these queries are simpler than those that one would expect to see in a production environment, they are able to illustrate from a more fundamental level the essential differences in performance between the three strategies. We look at joins in Section 4.3.

To explore the performance of the materialization strategies as a function of the selectivity of the query, we varied X across the entire shipdate domain and kept Y constant at 7 (96% selectivity). In other experiments (not presented in this paper) we varied Y and kept X constant and observed similar results (unless otherwise stated).

Additionally, at each point in this sample space, we varied the encoding of the LINENUM column among uncompressed, RLE, and bit-vector encodings (SHIPDATE was always RLE encoded). We experimented with the four different query plans described in Section 3.5: *EM-pipelined, EM-parallel, LM-pipelined, and LM-parallel*. Both LM strategies were implemented using the multi-column optimization.

Experiments were run on a Dell Optiplex GX620 DT with a 3.8 GHz Intel Pentium 4 processor 670 with HyperThreading, 2MB of cache, and a 800 Mhz FSB. The system had 4GB of main memory installed, of which 3.5GB were available to the operating system. The hard drive used was a 250GB Western Digital WD2500JS-75N.

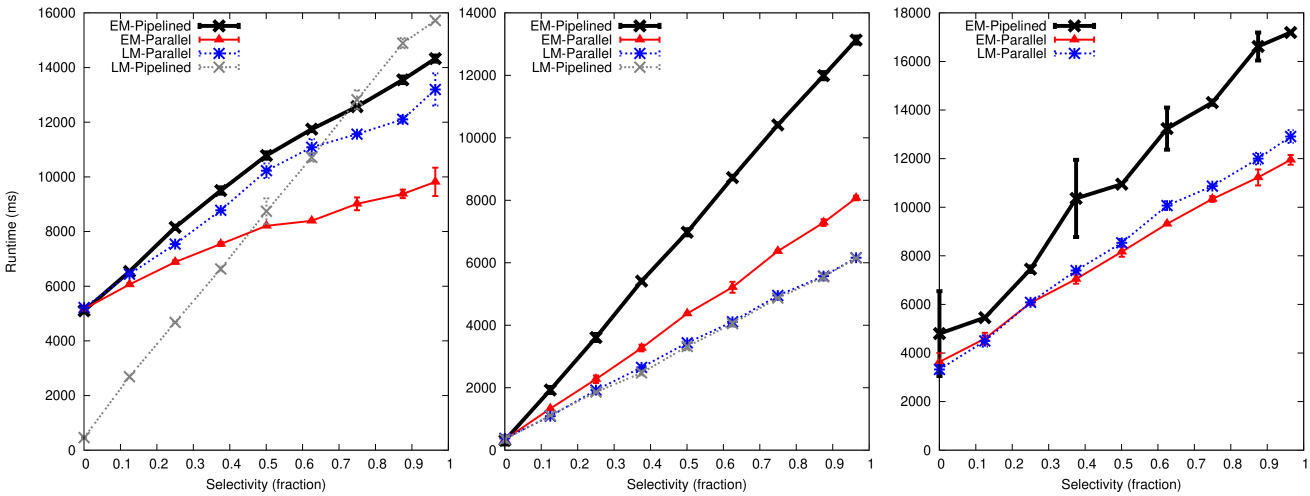
4.1 Experiment 1: Simple Selection Query

For this set of experiments, we consider the simple selection query presented both in Section 3.5 and in the introduction to this section above. Figure 11 (a), (b), and (c) shows the total end-to-end query time for the four materialization strategies when the LINENUM column stored uncompressed, RLE encoded, and bit-vector encoded respectively.

For the uncompressed LINENUM experiment (Figure 11 (a)), LM-pipelined is the clear winner at low selectivities. This is because the pipelined algorithm saves both I/O and CPU costs of reading in and applying the predicate to the large (250 MB) uncompressed LINENUM column since the first predicate is so selective and the matching tuples are so localized (since the SHIPDATE column is secondarily sorted) that entire LINENUM blocks can be skipped from being read in and processed. However at high selectivities LM-pipelined performs particularly poorly since the CPU cost of jumping to each matching position is more expensive than iterating through the block one position at a time if most positions will have to be jumped to. This additional CPU cost eventually dominates query time. At high selectivities, immediately making complete tuples at the bottom of the query plan (EM-parallel) is the best thing to do. EM-parallel consistently outperforms LM-parallel since the LINENUM predicate selectivity is so high (96%). In other experiments (not shown), we varied the LINENUM predicate across the LINENUM domain and observed that if both the LINENUM and the SHIPDATE predicate have medium selectivities, LM-parallel can beat EM-parallel (this is due to the LM advantage of waiting until the end of the query to construct tuples and thus it can avoid creating tuples that will ultimately not be output).

For the RLE-compressed LINENUM experiment (Figure 11 (b)), the I/O cost for all materialization strategies is negligible (the RLE encoded LINENUM column occupies only three 64k blocks on disk). At low query selectivities, the CPU cost is also minimal for all strategies and they all perform on the order of tens to hundreds of milliseconds. However, as the query selectivity increases, we observe the difference in costs of the strategies. Both EM strategies under-perform the LM strategies since tuples are constructed at the beginning of the query plan and tuple construction requires the RLE-compressed data to be decompressed (Section 2.1.2) precluding the performance advantages of operating directly on compressed data discussed in [3]. In fact the CPU cost of operating directly on compressed data is so small that the almost the entire query time for the LM strategies is the construction of the tuples and subsequent iteration over the results; hence both LM strategies perform similarly.

For the bit-vector compressed LINENUM experiment (Figure 11 (c)), we show the results for only three strategies since performing position filtering directly on bit-vector data (the DS3 operator) is not supported (since it is impossible to know in advance in which bit-string any particular position is located). Further, the advantage of reducing the number of values that the predicate has to be applied to for the LM-pipelined strategy is irrelevant for the bit-vector compression technique (since the predicate has already been applied a-priori as each bit-vector is a list of positions that match a particular value equality predicate



(a) (b) (c)

Figure 11: Run-times for four materialization strategies on selection queries with uncompressed (a), RLE compressed (b), and Bit-vector compressed (c) LINENUM column

- to apply a range predicate, the executor simply needs to OR together the relevant bit-vectors). Thus, only one LM algorithm is presented (LM-parallel).

Since there are only 7 unique LINENUM values, the bit-vector compressed column is a little less than 25% of the size on disk of the uncompressed column. Thus, the dominant cost of the query is the CPU cost of decompressing the bit-vector data which has to be done for both the LM and EM strategies. Hence, EM-parallel and LM-parallel perform similarly.

4.2 Experiment 2: Aggregation Queries

For this set of experiments, we consider adding an aggregation operator on top of the selection query plan (the full query is presented in the introduction to this section above). Figure 12 (a), (b), and (c) shows the total end-to-end query time for the four materialization strategies when the LINENUM column stored uncompressed, RLE encoded, and bit-vector encoded respectively.

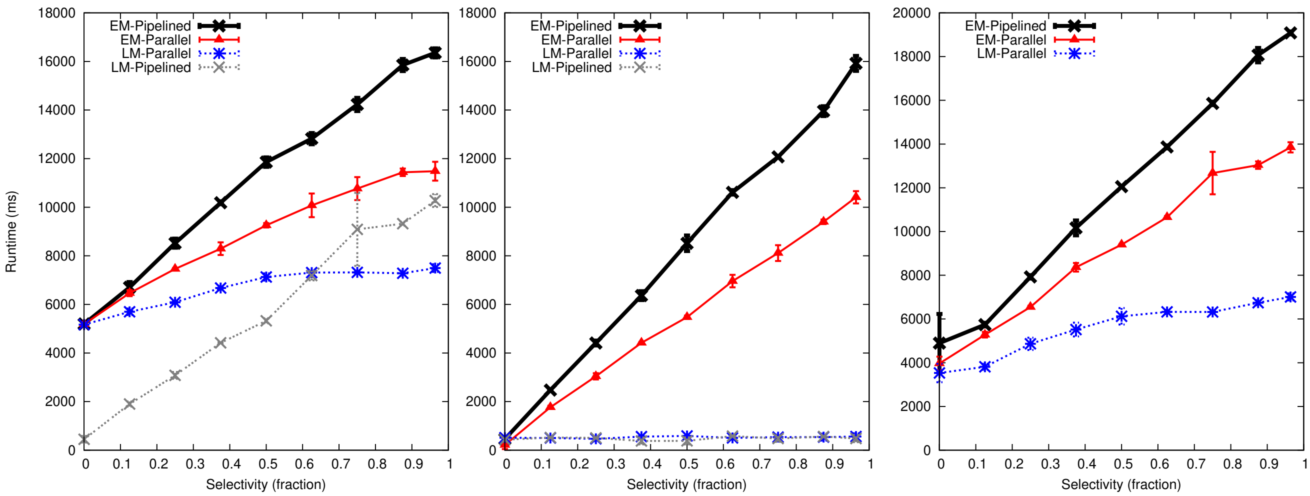
In each of these three graphs, the EM strategies perform similarly to their counterpart in Figure 11. This is because the CPU cost of iterating through the query results in Figure 11 is done by the aggregator (and the cost to iterate through the aggregated results and to perform the aggregation itself is minimal). However, the LM strategies all perform significantly better. In the uncompressed case (Figure 12(a)) this is because waiting to construct tuples is a big win since the aggregator significantly reduces the number of tuples that need to be constructed. For the compressed cases (Figure 12(a) and (b)) the aggregator iterator cost is also reduced since it can optimize its performance by operating directly on compressed data [3] so keeping data compressed as long as possible is also a win.

4.3 Joins

We now look at the effect of materialization strategy on join performance. If an early materialization strategy is used relative to a join, tuples have already been constructed before reaching the join operator, so the join functions as it would in a standard row-store system, with tuples being output. However, an alternative algorithm can be used if using a late materialization strategy. In this case, only columns that compose the join predicate are input to the join. The output of the join is set of pairs of positions in the two input relations for which the predicate succeeded. For example, the figure below shows the results of a join of a column of size 5 with a column of size 3.

$$\begin{array}{|c|}\hline 42\\ \hline 36\\ \hline 42\\ \hline 44\\ \hline 38\\ \hline\end{array} \bowtie \begin{array}{|c|}\hline 38\\ \hline 42\\ \hline 46\\ \hline\end{array} = \begin{array}{|c|c|}\hline 1&2\\ \hline 3&2\\ \hline 5&1\\ \hline\end{array}$$

For most join algorithms, the output positions for the left input relation will be sorted while the output positions of the right input relation will not. This is because the positions in the left column are usually iterated through in order, while the right relation is probed for join predicate matches. This asymmetric nature of join positional output implies that restricting other columns from the left input relation using the join output positions will be relatively fast (the standard merge join of positions can be used to extract column values); however, restricting other columns from the right input relation using the join output positions can be significantly more expensive (out of order posi-



(a) (b) (c)

Figure 12: Run-times for four materialization strategies on aggregation queries with uncompressed (a), RLE compressed (b), and Bit-vector compressed (c) LINENUM column

tions implies that a merge-join on position cannot be used to fetch column values).

Of course, a hybrid approach can be used where the right relation can send tuples to the join operator, while the left relation can send the single join predicate column. The join result is then a set of tuples from the right relation and an ordered set of positions from the left relation which can then be used to fetch values from relevant columns in the left relation to complete the tuple stitching process and create a single join result tuple. This allows for the advantage of only having to materialize values in the left relation for tuples that passed the join predicate without paying the penalty of an extra non-merge positional join with the right relation.

Multi-columns give joins another option as an alternative right (inner) table representation instead of forcing input tuples to be completely constructed or just sending the join predicate column. All relevant columns (columns that will be materialized after the join in addition to columns in the join predicate) are input to the join operator and as inner table values match the join predicate, the position of this value is extracted and used to get the appropriate value in all of the other relevant columns and the tuple constructed on the fly. This hybrid technique is useful if the join selectivity is low, and few tuples need be constructed.

To further examine the differences between these three materialization approaches for the inner table in a join operator (send just the unmaterialized join predicate column, send the unmaterialized relevant columns in a multi-column, or send materialized tuples), we ran a standard star schema join query on our TPC-H data between the orders table and customers table on customer key (customer key is a foreign key in the orders table and the primary key for the customers table), varying the selectivity of an input predicate on the orders table:

```
SELECT Orders.shipdate
       Customer.nationcode
FROM   Orders, Customer
WHERE  Orders.custkey=Customer.custkey
   AND Orders.custkey < X
```

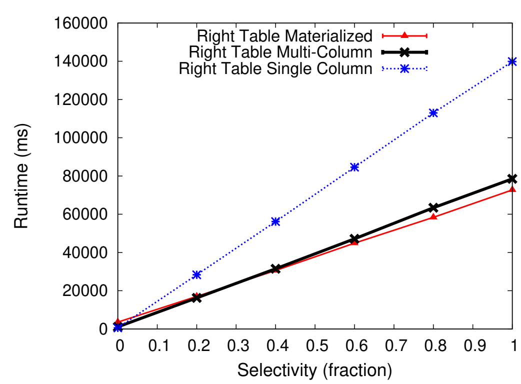


Figure 13: Run-times for three different materialization strategies for the inner table of a join query

Where X is varied so that a desired predicate selectivity can be acquired. For TPC-H scale 10 data, the orders table contains 15,000,000 tuples and the customer table 1,500,000 tuples and since this is a foreign key-primary key join, the join result will also have at most 15,000,000 tuples (the actual number is determined by the Orders predicate selectivity). The results of this experiment can be found in Figure 13.

Sending early materialized tuples and multi-column unmaterialized column data to the right-side input of the join operator results in similar performance numbers since the the multi-column advantage of only materializing relevant tuples is not helpful for a foreign key-primary key join of this type where there are exactly as many join results as there are join inputs. Sending just the join predicate column ("pure" late materialization) performs poorly due to the extra join of right-side positions. If the entire set of positions were not able to be kept in memory, late materialization would have performed even worse.

We do not present results for varying the materialization strategy of the left-side input table to the join operator since the issues at play are identical to the issues discovered in the previous experiments: if the join is highly selective or if the join results will be aggregated, a late materialization strategy should be used. Otherwise, EM-parallel should be used.

5 Related Work

The multi-column idea of combining chunks of columns covering the same position range together into one data structure is similar to the PAX [4] idea of taking a row-store page and splitting it into multiple *mini-pages* where each tuple attribute is stored contiguously. PAX does this to improve cache performance by maximizing inter-record spatial locality within a page. Multi-columns build on the PAX idea in the following ways: First, multi-columns are an in-memory data structure only and are created on the fly from different columns stored separately on disk (where pages for the different columns on disk do not necessarily match-up position-wise). Second, positions are first class citizens in multi-columns and may be accessed and processed separately from attribute values. Finally, mini-columns are kept compressed inside multi-columns in their native compression format throughout the query plan, encapsulated inside a specialized data structures that facilitate direct operation on compressed data.

To the best of our knowledge, this paper contains the only study of multiple tuple creation strategies in a column-oriented system. C-Store [16] used LM-parallel only (until we extended it with additional strategies). Published descriptions of Sybase IQ [11] seem to indicate that they also perform LM-parallel. Ongoing work by Halverson et. al. [8] and Harizopoulos et. al. [9] that further explores the trade-offs between row- and column-stores use early materialization approaches for the column-store they implemented (the former uses EM-parallel, the latter uses EM-pipelined). MonetDB/X100 [7] uses late materialization implemented using a similar multi-column approach; however their version of position descriptors (they call them *selection vectors*) are kept separately from column values and data is decompressed in the cache, precluding the potential performance benefits of operating directly on compressed data both on position descriptors and on column values.

Finally, we used the analytical model presented in [8] as a starting point for the analytical model we presented in Section 3. However, we extended their model since they look only at scanning costs of the leaf query plan operators whereas we model the costs for the entire query since the query plans for different materialization strategies have different costs.

6 Conclusion

The optimal point at which to perform tuple construction in a column-oriented database is not obvious. This paper provides a systematic evaluation of a variety of strategies for when tuple construction should occur. We showed that late materialization has many advantages, but potentially incurs additional costs due to re-processing disk blocks, and hence early materialization is sometimes preferable. A good heuristic to use is that if output data is aggregated, or if the query has low selectivity (highly selective predicates), or if input data is compressed using a light-weight compression technique, a late materialization strategy should be used. Otherwise, for high selectivity, non-aggregated, non-compressed data, early materialization should be used. Further, the right input table to a join should be materialized before (or during if a multi-column is input) the join operation. Using an analytical model to predict query performance can facilitate materialization strategy decision-making.

References

[1] http://www.addamark.com/products/sls.htm.

[2] C-store code release under bsd license. http://db.csail.mit.edu/projects/cstore/, 2005.

[3] D. J. Abadi, S. Madden, and M. Ferreira. Integrating compression and execution in column-oriented database systems. In *SIGMOD*, 2006.

[4] A. Ailamaki, D. J. DeWitt, M. D. Hill, and M. Skounakis. Weaving relations for cache performance. In *VLDB '01*, pages 169–180, San Francisco, CA, USA, 2001.

[5] P. Boncz. Monet: A next-generation dbms kernel for query-intensive applications. Phd thesis, Universiteit van Amsterdam, 2002.

[6] P. A. Boncz and M. L. Kersten. MIL primitives for querying a fragmented world. *VLDB Journal: Very Large Data Bases*, 8(2):101–119, 1999.

[7] P. A. Boncz, M. Zukowski, and N. Nes. Monetdb/x100: Hyper-pipelining query execution. In *CIDR*, pages 225–237, 2005.

[8] A. Halverson, J. Beckmann, and J. Naughton. A principled comparison of storage optimizations for row-store and column-store systems for read-mostly query workloads. in submission. http://www.cs.wisc.edu/~alanh/tr.pdf.

[9] S. Harizopoulos, V. Liang, D. Abadi, and S. Madden. Performance tradeoffs in read-optimized databases. In *VLDB*, page To appear, 2006.

[10] Kx Sytems, Inc. Faster database platforms for the real-time enterprise: How to get the speed you need to break through business intelligence bottlenecks in financial institutions. http://library.theserverside.com/data/document.do?res_id=1072792428_967, 2003.

[11] R. MacNicol and B. French. Sybase IQ multiplex - designed for analytics. In *VLDB*, pages 1227–1230, 2004.

[12] P. E. O'Neil and D. Quass. Improved query performance with variant indexes. In *Proceedings ACM SIGMOD '97 Tucson, Arizona, USA*, pages 38–49, 1997.

[13] R. Ramamurthy, D. Dewitt, and Q. Su. A case for fractured mirrors. In *VLDB*, 2002.

[14] S. Khoshafian, et al. A query processing strategy for the decomposed storage model. In *ICDE*, 1987.

[15] P. Selinger, M. Astrahan, D. Chamberlin, R. Lorie, and T. Price. Access path selection in a relational database management system. In *Proceedings of the ACM SIGMOD Conference on Management of Data*, pages 23–34, Boston, MA, 1979.

[16] M. Stonebraker, D. J. Abadi, A. Batkin, X. Chen, M. Cherniack, M. Ferreira, E. Lau, A. Lin, S. Madden, E. J. O'Neil, P. E. O'Neil, A. Rasin, N. Tran, and S. B. Zdonik. C-store: A column-oriented dbms. In *VLDB*, pages 553–564, 2005.