Materialization Strategies in a Column-Oriented DBMS

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

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

David J. DeWitt
UW Madison
dewitt@cs.wisc.edu

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

Materialization Strategies in a Column-Oriented DBMS

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

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

David J. DeWitt
UW Madison
dewitt@cs.wisc.edu

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

Abstract

There has been renewed interest in column-oriented database architectures in recent years. For read-mostly query workloads such as 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, however, they must present the same interface to client applications as do row stores, which implies that they 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, but the optimal point at which to do the conversion is not obvious. This problem can be considered as the opposite of the projection problem in row-store systems: while row-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 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 [8, 9, 12, 13, 14]. 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.

Column-stores are essentially a modification only to the physical data structures of a database: at the logical and view level, a column-store looks identical to a row-store. For this reason, most column-stores choose to offer a standards-compliant relational database interface (e.g., ODBC, JDBC, etc). As such, separate columns must ultimately be stitched together into tuples of data to be output. Determining when to do this stitching together 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 row-oriented databases (where projections are almost always performed as soon as an attribute is no longer needed) suggest a natural tuple construction policy: at each point at which 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. 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. 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(s) 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.

There is another strategy that can be more efficient, however. We call this second approach *late materialization*, because it does not 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 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 satisfy 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 small, highly-compressible data structures 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 with 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).

The main contribution of this paper is not to introduce new materialization strategies (as described in the related work, many of these strategies have been used in other column-stores). Rather, it is to systematically explore the trade-offs between different strategies and provide a foundation for choosing a strategy for a particular query. We focus on standard warehouse-style queries: read-only workloads with selections, aggregations, and joins. We extended the C-Store column-oriented DBMS [14] with a variety of materialization strategies, and experimentally evaluate the effects of varying selectivities, compression techniques, and query plans on these strategies. Further, we provide a 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.

The remainder of this paper is organized as follows. Section 2 gives a brief overview of the C-Store query executor. We illustrate the trade-offs between materialization strategies in Section 3 and then present both pseudocode and an analytical model for example query plans using each strategy in Section 4. We validate our models experimentally (using a version of C-Store we extended) in Section 5. Finally, we describe related work in Section 6 and conclude in Section 7.

2 The C-Store Query Executor

We chose to use C-Store as the column-oriented DBMS to extend and experiment with since we were already familiar with the source code. Since this is the system in which the various materialization strategies were implemented for this study, we now provide a brief overview of the relevant details of the C-Store query executor, which is more fully described in previous work [4, 14] and available in an open source release [2]. The components of the query executor most relevant 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 for manipulating 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 *data source*) 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. Additionally, all C-Store data sources accept predicates 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 as blocks of encoded data, wrapped inside a C++ object that provides iterator-style (hasNext() and getNext() methods) and vector-style [8] (asArray()) access to the data.

In Section 4.1 we give pseudocode for the C-Store operators relevant to this paper: DataSource (Select), AND, and Merge. These operators are used to construct the query plans we experiment with. We also describe the Join operator in Section 5.3. 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 lists of $(position, value)$ pairs into a single output list of n-attribute tuples.

3 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 [7, 8, 14] store columns in position order; i.e., to reconstruct the i'th tuple, one uses the i'th value from each column. This accelerates the tuple reconstruction process.

As described in the introduction, tuple reconstruction can occur at different points in a query plan. *Early materialization* (EM) constructs tuples as soon as (or sometimes before) tuple values are needed in the query plan. *Late*

materialization (LM) constructs tuples as late as possible, sometimes even at the query output. Each approach has a set of advantages.

3.1 Late Materialization Advantages

The primary advantages of late materialization are that it allows the executor to use high-performance operations on compressed, column-oriented data and to defer tuple construction to later in the query plan, possibly allowing it to construct fewer tuples.

3.1.1 Column-Oriented Data Structures

One 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 were used to store the column on disk. Previous work [4] has showed that techniques such as run length encoding (RLE) of column values and bit-vector encoding are well suited for column stores and can easily be operated on directly. For example, an entire run of RLE-encoded values can be processed in a single operator loop. Tuple construction requires decompressing RLE data, since generally repeats are of values in a single column, not entire tuples.

Second, looping through values from a column-oriented data structure tends to be much faster than looping through values using a tuple iterator interface. First, entire cache lines are filled with values from the same column. This maximizes the efficiency of the memory bandwidth bottleneck [5] as the cache prefetcher only fetches relevant data. Second, high IPC (instructions-per-cycle) vector processing code can be written for column block access by taking advantage of modern super-scalar CPUs [6, 7, 8].

3.1.2 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 into summary tuples. Thus, if the executer waits long enough before constructing a tuple, it might be able to avoid constructing it altogether.

3.2 Early Materialization Advantages

The fundamental problem with delaying tuple construction is that in some cases columns may need to be accessed multiple times in the query plan. For example, suppose a column is accessed once to retrieve positions matching a predicate and a second time (later in the plan) to retrieve its values. If the matching positions cannot be determined from an index, then the column values will be accessed twice. Assuming proper query pipelining, the reaccess will incur no disk costs (the disk block will be in the buffer cache), but there will be a CPU cost in scanning the block to extract the values corresponding to the given positions. This cost can be substantial if the positions are not in sorted order (e.g., they were reordered by a join, as discussed in Section 5.3).

In early materialization, as soon as a column is accessed, its values are added to an intermediate-result tuple, eliminating the need for future reaccesses. 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, constructing only relevant tuples, operating directly on column-oriented compressed data, and high value iteration speeds), if the column reaccess cost at tuple reconstruction time is high, a performance penalty is paid.

4 Query Processor Design

Having described the fundamental trade-offs between early and late materialization, we now present both pseudocode and an analytical model for component operators of each materialization strategy and give detailed examples of how these strategies are translated into query plans in a column-oriented system.

4.1 Operator 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 either vectors of positions or vectors of positions and values.
- AND operators that merge several position lists into a single position list in which positions are present only if they were present in all input position lists.
- Tuple construction operators that combine multiple narrow tuples of positions and values into wider tuples.

These operators are sufficient to express simple selection queries using each strategy. We omit the model of operators needed for more complex queries. We use the notation in Table 1 to describe the costs of the different operators.

4.2 Data Sources

In this section, we consider the cost of accessing a column on disk via a data source operator. We consider four cases (the first and third used by LM strategies, the second and fourth used by EM):

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.

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 (position, value) pairs.

$\|C_i\|$	Number of disk blocks in Col_i
$\|\|C_i\|\|$	Number of "tuples" in Col_i
$\|\|POSLIST\|\|$	Number 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	Avg. run-length in RLE encoded columns (RL_c) or position lists (RL_p) (equal to one if uncompressed)

Table 1. Notation used in analytical model

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

$$CPU =$$

$$|C_i| * BIC + \quad (1)$$

$$||C_i|| * (TIC_{COL} + FC)/RL_c + \quad (3,4)$$

$$SF * ||C_i|| * FC \quad (5)$$

$$IO = (\frac{|C_i|}{PF} * SEEK + |C_i| * READ) * (1 - F) \quad (2)$$

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

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.

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 position pos in the column and applies a predicate with selectivity SF. Tuples that satisfy the predicate are merged with EM_i to create tuples of the form $(pos, < a_1, \ldots, a_n, a_{n+1} >)$ that contain 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.

4.3 Multicolumn AND

The AND operator takes in k position lists, $inpos_1 \ldots inpos_k$ and produces a new list of positions representing the intersection of these input lists, $outpos$. Since operating directly on positions is fast, the cost of the AND operator in query plans tends to be insignificant relative to the other operators. We present the AND analysis in our full technical report [3] and omit it here to save space.

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

$$CPU =$$

$$|C_i| * BIC + \quad (1)$$

$$||POSLIST||/RL_p * (TIC_{COL}) + \quad (3)$$

$$||POSLIST||/RL_p * (TIC_{COL} + FC) \quad (4)$$

$$IO = (\frac{|C_i|}{PF} * SEEK + SF * |C_i| * READ) * (1 - F) \quad (2)$$

/* F=1 and IO → 0 if col already accessed */

/* $SF * |C_i|$ is a lower bound for the blocks needed to be read in. For highly localized data (like the semi-sorted data we will work with), this is a good approximation*/

Figure 2: Pseudocode and cost formulas DS-Case 3.

```
DS_Scan-Case4(Column C, Pred p, Table EM)
 1. for each block b in C
 2. read b from disk (if necessary)
 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>
```

$$CPU =$$

$$|C_i| * BIC + \quad (1)$$

$$||EM_i|| * TIC_{TUP} + \quad (3)$$

$$||EM_i|| * ((FC + TIC_{TUP}) + FC) \quad (4)$$

$$SF * ||EM_i|| * (TIC_{TUP}) \quad (5)$$

$$IO = (\frac{|C_i|}{PF} * SEEK + SF * |C_i| * READ) * (1 - F) \quad (2)$$

/* $SF * |C_i|$ is a lower bound for the blocks needed to be read in, as in Figure 2 */

Figure 3: Pseudocode and cost formulas for DS-Case 4.

4.4 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 4. The analysis assumes the k sets of values are resident in main memory, since they are produced by a child operator, and that each set has the same cardinality.

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 5.

4.5 Example Query Plans

The use of the above presented operators is illustrated in Figures 6 and 7 for the query:

```
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 4: Pseudocode and cost formulas for Merge.

```
SPC(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.     call Merge sub-routine
 5.     check predictates
 6.     output matching tuples
```

$CPU =$

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

$$Merge(c1, ..., ck)+ \quad (4)$$

$$||C_i|| * FC * \prod_{j=1...(i-1)} (SF_j)+ \quad (5)$$

$$||C_k|| * FC * \prod_{j=1...k} (SF_j)+ \quad (6)$$

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

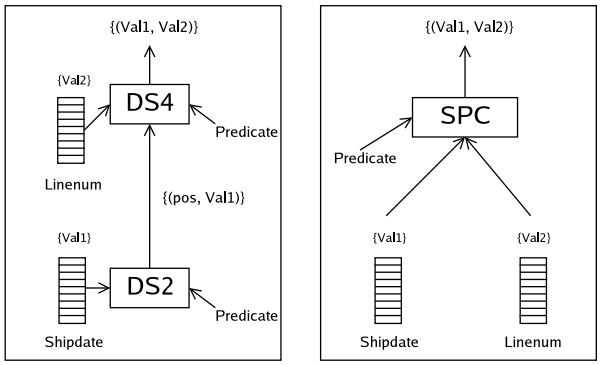
Figure 5: Pseudocode and cost formulas for SPC.

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

where lineitem is a table taken from TPC-H [1], a benchmark that models data typically found in decision support and data warehousing applications.

One EM query plan, shown in Figure 6(a), uses a DS2 operator (Data Scan Case 2) operator to scan the shipdate 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 $(shipdate, linenum)$ result tuples.

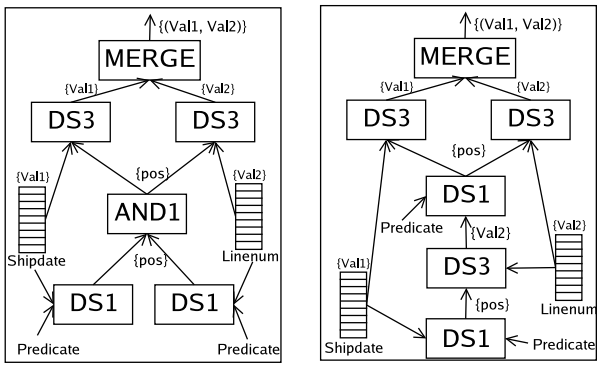
Another possible EM query plan, shown in Figure 6(b), constructs tuples at the very beginning of the plan - merging all needed columns at the leaf node as it applies the predicates using a SPC operator. The key difference between these early materialization strategies is that while the latter strategy has to scan and process all blocks for all input columns, the former plan applies each predicate in turn and constructs tuples 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; EM-pipelined is likely better if there are highly selective predicates.



(a) (b)

Figure 6: Query plans for EM-pipelined (a) and EM-parallel (b) strategies. DS2 is shorthand for DS_Scan-Case2. (Similarly for DS4).

As in EM, there are both pipelined and parallel late materialization strategies. LM-parallel is shown in Figure 7(a) and LM-pipelined is shown in Figure 7(b). LM-parallel begins with two DS1 operators, one for the shipdate 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 shipdate and linenum columns. As these values are obtained they are streamed into a merge operator to produce a stream of (shipdate, linenum) result tuples.



(a) (b)

Figure 7: Query plans for LM-parallel (a) and LM-pipelined (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 side effect, the need for the AND operator is eliminated.

4.6 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 they 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 of corresponding values for a specified position range of a particular attribute (MonetDB [8] calls this a *vector*, PAX [5] 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.

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

An AND operator takes two multi-columns with overlapping covering position ranges and creates a new multi-column where the covering position range and position descriptor are 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 ANDing normal position lists. The only difference is that in addition to performing the intersection of the position lists, ANDing multi-columns requires copying pointers to mini-columns to the output multi-column, but this is a low 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-column blocks are worked on in each operator iteration, so that column-subsets can be pipelined up the query tree. With this optimization, there is no DS3 I/O cost for a reaccessed column.

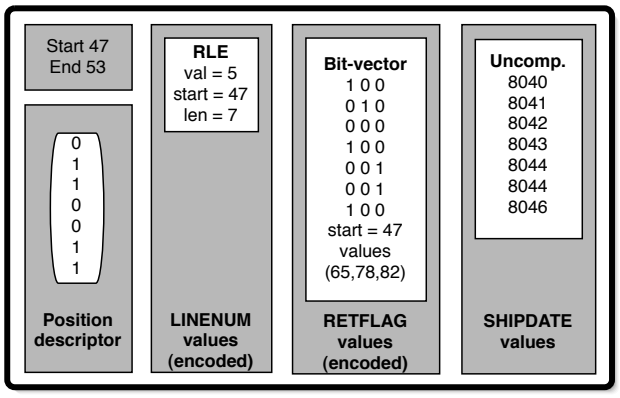


Figure 8: An example multi-column block containing values for the SHIPDATE, RETFLAG, and LINENUM columns. The block spans positions 47 to 53; within this range, positions 48, 49, 52, and 53 are active.

4.7 Predicted versus Actual Behavior

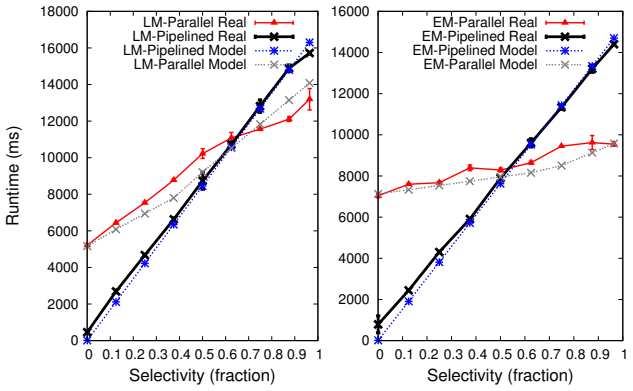
To gauge the accuracy of the analytical model, we compared the predicted execution time for the selection query (1) in Section 4.5 above with the actual execution time obtained using the C-Store prototype and a TPC-H scale 10 version of the lineitem projection primarily and secondarily sorted by $returnflag$ and $shipdate$, respectively.

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

The results obtained are presented in Figures 9(a) and 9(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. For these results, we encoded the shipdate column (into one block) using RLE encoding and kept the linenum column uncompressed (the 60,000,000 linenum tuples occupy 3696 64KB blocks). Table 2 contains the constant values used by the analytical model. These constants were obtained from microbenchmarks on the C-Store system; they were not reverse-engineered to make the model 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 ($numOutTuples * TIC_{TUP}$). We assume a two thirds overlap of I/O and CPU (which is what we have generally found when running C-Store on our testbed machines).

Here, the important observation is that the models' predictions are quite accurate (at least for this query), which helps validate our understanding of the two strategies. The actual results will be further discussed in Section 5. Additionally, we tested our model on several other cases, including the same query presented here but using an RLE-compressed linenum column (occupying only 7 disk blocks) as well as additional queries in which both the shipdate and linenum predicates were varied. We consistently found the model to reasonably predict our experimental results.



(a) (b)

Figure 9: Predicted and observed performance for late (a) and early (b) materialization strategies on selection queries.

5 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. 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 the selection query from Section 4.5:

```
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, their simplicity aids in distilling the essential differences in performance between the materialization strategies. We consider joins separately in Section 5.3.

To explore the performance of the 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 4.5: *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 Hyper-Threading, 2MB of cache, and a 800 MHz FSB. The system had 4GB of main memory installed, of which 3.5GB were available to the database. The hard drive used was a 250GB Western Digital WD2500JS-75N.

5.1 Simple Selection Query

For this set of experiments, we consider the simple selection query presented both in Section 4.5 and in the intro-

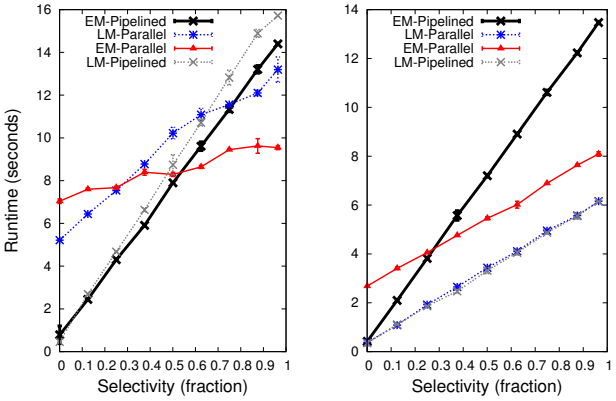


Figure 10: Run-times for four materialization strategies on selection queries with uncompressed (a) and RLE compressed (b) LINENUM column.

duction to this section above. Figures 10 (a) and (b) show the total end-to-end query time for the four materialization strategies when the LINENUM column is stored uncompressed and RLE encoded, respectively.

For the uncompressed LINENUM experiment (Figure 10 (a)), the pipelined strategies are the clear winners at low selectivities. The pipelined algorithm reduces the I/O and CPU costs of reading in and applying the predicate to the large (250 MB) uncompressed LINENUM column because the first predicate is highly selective and the matching tuples are sufficiently localized (since the SHIPDATE column is secondarily sorted) that entire LINENUM blocks can be completely skipped. At high selectivities, however, pipelined strategies perform poorly since the CPU cost of jumping to each matching position is more expensive than simply iterating through the block one position at a time, and 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 option; almost all tuples will need to be materialized, and EM-parallel has the lowest per-tuple construction cost of any of the strategies.

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 10 (b)), the I/O cost for all materialization strategies is negligible (the RLE encoded LINENUM column occupies only seven 64k blocks on disk). At low query selectivities, the CPU cost is also low for all strategies. However, as the query selectivity increases, we observe the difference in costs of the strategies. Both EM strategies underperform 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 3.1.1), precluding the performance advantages of operating directly on compressed data discussed in [4]. In fact, the CPU cost of operating directly on compressed data is so small that almost the entire query time for the LM strategies is the construction of tuples and subsequent iteration over the results; hence both LM strategies perform similarly.

We also ran experiments when the LINENUM column column was bit-vector compressed. The dominant cost factor for these sets of experiments was decompression, so EM and LM performed similarly. The graphs and analysis of these experiments are presented in our technical report [3].

5.2 Aggregation Queries

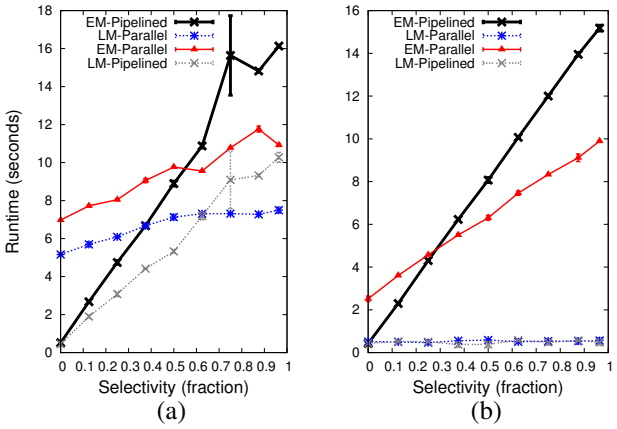


Figure 11: Run-times for four materialization strategies on aggregation queries with uncompressed (a) and RLE compressed (b) LINENUM column.

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 11 (a) and (b) shows the total end-to-end query time for the four materialization strategies when the LINENUM column is stored uncompressed and RLE encoded, respectively.

In each of these graphs, the EM strategies perform similarly to their counterparts in Figure 10: the CPU cost of producing and iterating through the early-materialized tuples easily dominates the cost of producing and iterating through the aggregate result. By contrast, the LM strategies all perform significantly better than before. In both the uncompressed and compressed cases (Figure 11(a) and (b), respectively), delaying tuple construction allows the plan to construct only those few tuples produced as output by the aggregator and to skip constructing the tuples corresponding to its inputs. Additionally, in the compressed case, the cost of aggregation is also reduced because the aggregator is able to operate extremely efficiently on compressed

data [4].

5.3 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 and outputs tuples. An alternative algorithm can be used with a late materialization strategy, however. In this case, only the columns that compose the join predicate are input to the join. The output of the join is a 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.

42						
36						
42	⋈	38	=	1	2	
44		42		3	2	
38		46		5	1	

For many join algorithms, the output positions for the left (outer) input relation will be sorted while the output positions of the right (inner) 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, since the standard merge join of positions can be used to extract column values. Restricting other columns from the right input relation using the join output positions can be significantly more expensive, however, as the out-of-order positions preclude the use of a merge-join on position to retrieve column values.

Of course, a hybrid approach could be used in which the right relation sends tuples to the join operator while the left relation sends only the single join predicate column. The join result would then be a set of tuples from the right relation and an ordered set of positions from the left relation; the positions from the left relation could easily be used to retrieve additional columns from that relation and complete the tuple construction process. This approach has the advantage of only materializing values in the left relation corresponding to tuples that pass the join predicate while avoiding the penalty of materializing values from the right relation using unordered positions.

Multi-columns provide another option for the representation of the right (inner) relations. All relevant columns (i.e., columns to be materialized after the join plus the predicate column) are input to the join operator as a multi-column. As inner table values match the join predicate, the position of the value is used to retrieve the values for other columns, and tuples are constructed on the fly. This hybrid technique is useful when the join selectivity is low and few tuples need to be constructed, but is otherwise expensive, since it potentially requires a particular tuple from the inner relation to be constructed multiple times.

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 the customers table on customer key (customer key is a foreign key in the orders table and the primary key for the customers table), where the less-than predicate on customer key is varied to obtain the desired selectivity:

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

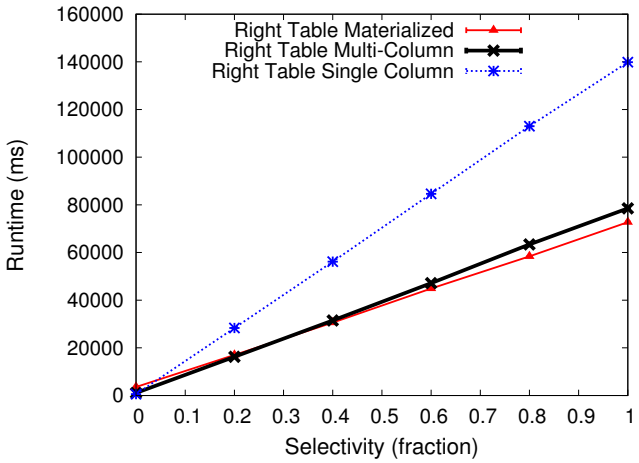


Figure 12: Run-times for three different materialization strategies for the inner table of a join query. Late materialization is used for the outer table.

For TPC-H scale 10 data, the orders table contains 15,000,000 tuples and the customer table 1,500,000 tuples. 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 12. Sending either early materialized tuples or multi-columns as the right-side input of the join operator results in similar performance, as the multi-column advantage of only materializing relevant tuples is not helpful for a foreign key-primary key join where there are exactly as many join results as join inputs. Sending just the join predicate column performs poorly due to the overhead of subsequent materialization using unordered positions. If the entire set of positions were not able to be kept in memory, late materialization would have performed even more poorly.

We do not present results for varying the materialization strategy of the left-side input table to the join operator since the trade-offs are identical to those discussed in 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.

6 Related Work

To the best of our knowledge, this paper contains the only study of multiple tuple creation strategies in a column-oriented system. C-Store [14] used LM-parallel only (until we extended it with additional strategies). Published descriptions of Sybase IQ [12] seem to indicate that it also uses LM-parallel. Papers by Halverson et al. [10] and Harizopoulos et al. [11] that further explore 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 [8] uses late materialization implemented using a similar multi-column approach; however, their version of position descriptors (they call them *selection vectors*) is 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. Thus, previous work has tended to choose a materialization strategy a priori without justification and has not examined trade-offs between these choices.

The multi-column optimization of combining chunks of columns covering the same position range together into one data structure is similar to the PAX [5] 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 PAX 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 in specialized data structures that facilitate direct operation on compressed data.

7 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. We are also optimistic that our analytical model can be used to predict query performance and help choose a materialization strategy at query planning and optimization time.

8 Acknowledgements

We would like to thank the C-Store team for their helpful feedback and ideas. This work was supported by the National Science Foundation under grants IIS-048124, CNS-0520032, IIS-0325703 and an NSF Graduate Research Fellowship.

References

[1] TPC-H. `http://www.tpc.org/tpch/`.

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

[3] D. Abadi, D. Myers, D. DeWitt, and S. Madden. Materialization strategies in a column-oriented dbms. MIT CSAIL Technical Report. MIT-CSAIL-TR-2006-078.

[4] D. J. Abadi, S. Madden, and M. Ferreira. Integrating compression and execution in column-oriented database systems. In *SIGMOD*, pages 671–682, 2006.

[5] 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.

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

[7] 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.

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

[9] G. Copeland and S. Khoshafian. A decomposition storage model. In *SIGMOD*, pages 268–279, 1985.

[10] A. Halverson, J. Beckmann, and J. Naughton. A comparison of c-store and row-store in a common framework. Technical Report, UW Madison Department of CS, TR1566, 2006.

[11] S. Harizopoulos, V. Liang, D. J. Abadi, and S. Madden. Performance tradeoffs in read-optimized databases. In *VLDB*, pages 487–498, 2006.

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

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

[14] 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.