A Comparison of C-Store and Row-Store in a Common Framework

Alan Halverson Jennifer L. Beckmann Jeffrey F. Naughton David J. DeWitt

University of Wisconsin-Madison
1210 W. Dayton St.
Madison, Wisconsin 53706 USA
{alanh,jbeckmann,naughton,dewitt}@cs.wisc.edu

Paper Id: 355

Abstract

Recently, a "column store" system called C-Store has shown significant performance benefits by utilizing storage optimizations for a read-mostly query workload. The authors of the C-Store paper compared their optimized column store to a commercial row store RDBMS that is optimized for a mixture of reads and writes, which obscures the relative benefits of row and column stores. In this paper, we describe two storage optimizations for a row store architecture given a read-mostly query workload – "super tuples" and "column abstraction." We implement both our optimized row store and C-Store in a common framework in order to perform an "apples-to-apples" comparison of the optimizations in isolation and combination. We also develop a detailed cost model for sequential scans to break down time spent into three categories – disk I/O, iteration cost, and local tuple reconstruction cost. We conclude that, while the C-Store system offers tremendous performance benefits for scanning a small fraction of columns from a table, our optimized row store provides disk storage savings, reduced sequential scan times, and low additional CPU overheads while requiring only evolutionary changes to a standard row store.

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the VLDB copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Very Large Data Base Endowment. To copy otherwise, or to republish, requires a fee and/or special permission from the Endowment.

Proceedings of the 32nd VLDB Conference, Seoul, Korea, 2006

1 Introduction

Recently, a column-oriented storage system called C-Store [7] has shown provocative performance results when compared to a commercial row-oriented DBMS. Their comparison of the read-optimized C-Store ideas to a write-optimized commercial DBMS obscures the relative benefits of row and column storage for read-mostly workloads. For example, one sequential scan query in the C-Store evaluation takes 2.54 seconds for C-Store while the DBMS takes 18.47 seconds – even with a materialized view that directly answers the query. In this paper, we show that the row store can also be optimized for a read-mostly query workload, and the query above can be run in as little as 1.42 seconds with our optimized row store. In an attempt to shed light on the comparison between the two, we implement both a read-optimized row store and the C-Store system in a common framework.

The C-Store architecture uses several main techniques to improve performance when compared to current commercial relational systems. First, C-Store stores each column of a relation separately on disk. In this way, scanning a small fraction of the columns from a relation with many columns saves disk I/O. Second, it carefully packs column values into large page-sized blocks to avoid per-value overheads. Third, C-Store uses a combination of sorting and value compression techniques to further reduce disk storage requirements. Both the page packing and sorting/compression techniques are an attempt to trade decreased I/O for increased CPU utilization.

The performance evaluation presented in the C-Store paper uses a modified TPC-H [4] schema and query workload to measure the combined effects of their performance improvement techniques. The reported results are very impressive – the C-Store system provides a significant performance improvement compared with a commercial row store. Although the com-

mercial row store compares poorly, an optimized row store can benefit from most of the same performance techinques proposed for the C-Store system. Specifically, our optimized row store uses both careful page packing, which we call "super tuples," and sorting to enable compression, which we call "column abstraction." The remaining technique – column storage – is the primary difference between row- and column-oriented storage. Careful page packing is particularly low-hanging fruit for a row store. Enforced sorting of the relation and storing repeating values only once to save space is slightly more effort, as it breaks the one-to-one mapping of the logical relational schema to the physical tuple on disk.

The main contributions of our paper are as follows:

- We provide descriptions of the "super tuple" and "column abstraction" performance techniques to optimize for a read-mostly query workload.

- We build a software artifact to evaluate these performance improvements for both the row and column stores in isolation and in combination, using a common storage manager. Our experiments vary tuple width, number of rows, level of sorting and column abstraction, and number of columns scanned to identify performance trends.

- We propose and validate a formal cost model for sequential scan for both row and column storage. The model takes into account the storage improvements and their effects on performance by identifying three factors which contribute to overall scan time. We compare the model predictions with our experimental results. We also use the model to forecast the behavior of systems and scenarios to gain further insight into performance trends.

The rest of the paper proceeds as follows. In Section 2, we present the storage optimizations and implementation details. Section 3 describes our experimental prototype and evaluates the storage optimizations in isolation and combination to discover performance trends. We then develop our formal cost model, with validation and forecasting, in Section 4. We describe related work in Section 5, and offer conclusions and future directions for research in Section 6.

2 Storage Optimizations

In this section, we describe the "super tuple" and "column abstraction" optimizations for the row store architecture. To illustrate the effects of each storage option, we will use an instance of a materialized view defined in the C-Store [7] paper. The view is based on a simplified version of the schema from the TPC-H benchmark [4], and is defined using SQL as follows:

L_RETURNFLAG	C_NATIONKEY	L_EXTENDEDPRICE
A	3	23
A	3	34
A	9	64
N	3	88
N	14	49
R	9	16
R	9	53
R	9	7
R	11	63
R	21	72
R	21	72

Table 1: **Example instance of materialized view D4**

```
CREATE VIEW D4 AS
SELECT L_RETURNFLAG, C_NATIONKEY, L_EXTENDEDPRICE
FROM   Customer, Orders, Lineitem
WHERE   C_CUSTID = O_CUSTID
 AND O_ORDERID = L_ORDERID
ORDER BY   L_RETURNFLAG, C_NATIONKEY;
```

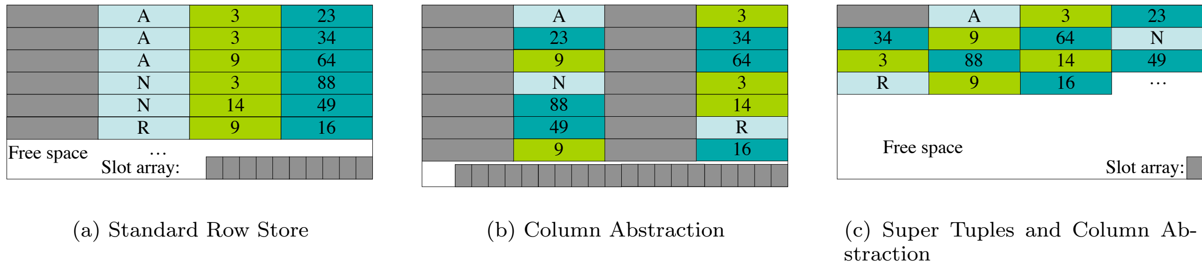
The definition is identical to the view called D4 in the C-Store paper with the exception of the secondary `ORDER BY` on the `C_NATIONKEY` column. Table 1 contains an instance of the D4 view that we use in all examples for this section. Figure 1(a) shows how a standard row store would layout the first few rows of the D4 view on a disk page.

2.1 Super Tuples

All of the major DBMS products use a variant of the slotted page for storage of tuples in a table. Slotted pages use an array of slots that point to the actual tuples within the page. Typically each tuple is prefaced by a header that provides metadata about the tuple. For example, metadata in the Shore storage manager [2] includes the type of tuple (small or large), the size of the user-specified record header, and the total size of the record if it is larger than one page and split across disk pages. The tuple header is implementation specific, but typically is 8-16 bytes in addition to the tuple's slot entry.

While the slotted page design provides a generic platform for a wide range of data storage needs, these per-tuple overheads can be problematic. Even for an 80 byte tuple, a 16 byte overhead is 20%. We reduce per-tuple overhead by packing many tuples into page-sized "super tuples." For fixed-length tuples, the super tuple is an array of tuple-sized entries which can be indexed directly. For variable length tuples, the tuple length must be stored. The super tuple design uses a nested iteration model, which ultimately reduces CPU overhead and disk I/O.

An important side effect of using super tuples is that external addressability of individual tuples is more difficult. Both the C-Store design and our optimized row store trade the storage benefits derived from tight packing of values for additional overhead associated with utilizing and maintaining value indexes.



(a) Standard Row Store

(b) Column Abstraction

(c) Super Tuples and Column Abstraction

Figure 1: **Row layout on storage pages for view D4 for (a) the standard row store, (b) row store with column abstraction, and (c) row store with super tuples and column abstraction**

L_RETURNFLAG	C_NATIONKEY	L_EXTENDEDPRICE
A	***3***	***23***
A	3	***34***
A	***9***	***64***
N	***3***	***88***
N	***14***	***49***
R	***9***	***16***
R	9	***53***
R	9	***7***
R	***11***	***63***
R	***21***	***72***
R	21	***72***

Table 2: **Column abstraction encoding of data from Table 1. Only need to store values in boxes – other values are implicit.**

2.2 Column Abstraction

Sorting provides an opportunity for disk storage savings. If the database can guarantee that tuples are retrieved from storage according to the sort order, we can store each unique value in the sort column once and then store the remaining unsorted attributes separately, according to the specific storage architecture. In this paper, we use the term "column abstraction" to describe the process of storing repeating values once. Disk space savings are higher when the number of unique values in the sorted column is smaller.

The columns in a materialized view may come from different tables and be related to each other by one or more join keys. For example, consider the one-to-many relationship between the `C_NATIONKEY` and `L_EXTENDEDPRICE` columns in our example D4 materialized view. Even when D4 is sorted by `L_RETURNFLAG` first, we can save space on disk by storing `C_NATIONKEY` once for each related `L_EXTENDEDPRICE`. We show in Table 2 how the sort column(s) for view D4 can be used to more efficiently encode the same data. We show which values must be stored on disk by drawing boxes around them. `L_RETURNFLAG` and `C_NATIONKEY` are sort attributes for D4 which allows us to store repeating values for each attribute only once. Note that `C_NATIONKEY` is sorted only within each unique `L_RETURNFLAG` value, so we must store values such as 3 and 9 more than once. Figure 1(b) shows how we layout pages in our optimized row store using column abstraction for view D4. Note that storage needs have increased due to additional row headers for the abstracted columns. In Figure 1(c), we show that combining super tuples with column abstraction creates a more efficient disk page layout.

In general, a view may not specify an explicit sort. However, referential integrity constraints may specify an enforced one-to-many relationship between two or more tables in the view definition. We use the referential integrity information to insert an implicit sort on the columns from the one side of the one-to-many join(s). It is sufficent to sort on the primary key of the "one" side of the join. If the view does not project the key, we sort by all columns mentioned on the one side of the join. As an example, consider our instance of view D4. With enforced one-to-many relationships for Customer to Orders and Orders to Lineitem, we add a secondary sort on the `C_NATIONKEY` column when the view is not already sorted by that column. This implicit sort opens up another opportunity to store the repeating column(s) only once to save space. Sorting must be performed only once during population of the materialized view. At query runtime, scanning the view produces tuples in the correct sort order without additional sorting.

2.3 Updates and Indexing

Both C-Store and our optimized row store pose problems for updates and indexing. This is a result of a deliberate decision to optimize for scan-mostly read-mostly workloads. Our goal in this section is not to prove that our optimized row store can be efficiently updated, but rather, to mention that data in row-stores with our optimizations can be updated and indexed, although the performance of these operations will not match their counterparts in a standard row-store.

The super tuple and column abstraction optimizations create additional inconvenience in processing updates for both C-Store and row stores. Inserting rows may force super tuples to be rebuilt or split across two pages. Updates to existing rows may force several

rows in the table to be deleted and reinserted elsewhere. C-Store takes a "snapshot isolation" approach to handling updates in batch, and a similar technique can be used in our optimized row store.

Indexing columns in tables optimized for read-mostly also presents implementation challenges. C-Store only allows indexes on the primary sort column of each "projection". Their design allows updates to the index to be bounded to a specific range of values in the index, as the values and pages containing those values are correlated by the sort. Indexes on other columns of the table are possible for both C-Store and the optimized row store, but maintenance is expensive when table records move within a super tuple or are split to a new page due to inserts.

3 Evaluation

To evaluate the performance benefits of specific storage improvements, we created an experimental prototype. The prototype is designed to allow each storage optimization introduced in Section 2 to be applied in isolation and in combination for both the row and column stores. We report results for the column store only with the "super tuple" optimization, since the per-value overheads are several times larger than the data itself without super tuples.

We first provide a detailed description of the prototype. To calibrate the performance of our C-Store implementation, we then compare our implementation to the C-Store system [7] using query Q7 from the C-Store paper. Later in this section, we evaluate the benefits of the "super tuple" optimization for the row store, sorting and run-length encoding benefits for both the row and column stores, and finally the effects of combining the optimizations. We focus on identifying performance trends that emerge rather than trying to choose the "best" combination.

3.1 Experiments Description

We implemented the row store and column store architectures in a single prototype using Shore [2] as the storage manager. We implemented a sequential scan operator for the row and column stores that can operate over the super tuple and column abstraction optimizations. We ran the experiments on a dual processor Pentium 4 2.4GHz Xeon machine with 1GB of main memory running Fedora Core 3 with a stock 2.6.13 kernel. We created a hardware RAID-0 volume using six 250GB disk drives to contain the data volumes. A separate 250GB disk stored the system catalog information. Shore was configured to use a 32KB page size and a 512MB buffer pool. All reported results are the average of five runs.

By implementing all storage architectures and optimizations in a single prototype, our goal is to hold performance variables constant while changing only

a1	a2	a3	a4
1	*1*	*1*	*1*
1	1	*2*	*2*
1	1	*3*	*3*
1	*2*	*1*	*1*
1	2	*2*	*2*
1	2	*3*	*3*
1	*3*	*1*	*1*
...			
1	4	*3*	*3*
2	*1*	*1*	*1*
2	1	*2*	*2*
...			

Table 3: **Instance of Gen_2_4_3 table with boxes around actual values stored. Sorted by column a1 and a2**

the variable of interest. Our prototype avoids memory copies from the buffer pool whenever possible. Shore offers direct read-only access to data which allows us to minimize expensive copy-out operations.

In our C-Store implementation, we allocate 256MB in main memory to be divided equally among the columns scanned for sequential prefetching of pages. For example, when scanning 8 columns, we sequentially read 32MB from each column during the scan. Without prefetching, random I/O can easily dominate scan times for a column store when reading a large number of columns. The necessity for page prefetching in a column store is further motivated in [5].

We turned off locking and logging to match the settings used in the C-Store evaluation [7]. We believe this is fair since an underlying assumption of both papers is a read-mostly query workload and all queries being evaluated are read-only. We gathered results for a cold Shore buffer pool and file system cache. We ran our experiments with warm buffers as well, but do not report these results since the contribution of disk I/O to the total scan times does not change our analysis of performance trends. To eliminate file system caching effects, we unmounted and remounted the data volume just before each cold run.

All data sets consist of rows of 4, 8, 16, and 32 integer columns with a varying number of rows per data set. We synthetically generated the data to enable exploration of various column abstraction choices. The data for each column is a simple sequence of integers, starting at 1. When a new level of column abstraction starts, the column values at each lower level of abstraction reset and begin counting from 1 again. See Table 3 for an example. The frequency of each value within a column is important for column abstraction, but the exact values do not matter.

To evaluate the effects of sorting and encoding techniques on sequential scan performance, we generated data sets which provide encoding opportunities. Con-

sider the 4-column data set in Table 3. The rows are sorted first by column a1 and then by column a2. We call this data set Gen_2_4_3, and it contains 24 rows in total. Recall from Section 2.2 that column abstraction is the process of storing repeating values from sort column only once to save disk space. For the data set in Table 3, we have 2 unique values in column a1 and 4 unique values in column a2. For each unique a2 value, we have 3 unique values for columns a3 and a4. We have drawn boxes around the values in the data set that must be stored when using column abstraction. We use the name of the relation to describe the number of unique values at each level of column abstraction. In this case, the name Gen_2_4_3 specifies three levels, and specifies $2 * 4 * 3 = 24$ tuples. Our experimental data sets follow the same naming convention. The chosen data sets allowed us to measure the effects of both constant rows and constant total data size for all tuple widths.

3.2 C-Store Query 7

To ensure that our implementation of C-Store had performance representative of the system presented in [7], we acquired their code [8] and compared the performance of their implementation of a column store with ours on our hardware. The result was that our implementation of a column store was comparable to theirs. We present one representative query as an example of the comparison. We ran query Q7 from their evaluation on our benchmark hardware to establish a baseline. We also implemented query Q7 in our Shore-based prototype, which is represented in SQL:

```
SELECT c_nationkey, sum(l_extendedprice)
FROM lineitem, orders, customer
WHERE l_orderkey=o_orderkey AND
 o_custkey=c_custkey AND
 l_returnflag='R'
GROUP BY c_nationkey;
```

We loaded their D4 projection (materialized view) and implemented the query plan according to the method used by the C-Store system. We executed the query in our system and theirs using our hardware. The hardcoded query plan for Q7 in the C-Store prototype system assumes that the view is sorted by the L_RETURNFLAG column, and that the L_RETURNFLAG column is run-length encoded. We ran the query in the C-Store prototype on our benchmark hardware, and it took 4.67s. By contract, our Shore-based C-Store implementation took 3.95s for the same query plan, which provides evidence that our C-Store implementation does not introduce overheads that would render the rest of our experiments suspect. For comparison, the C-Store paper [7] reported a time of 2.54s for their system for query Q7 on their 3.0 GHz benchmark machine.

3.3 Super Tuple Effects

To show the benefits of the super tuple storage optimization, we performed two experiments. First, we measured the effects of varying the number of columns per tuple scanned when combined with the super tuple optimization. We then compared standard and super tuple row storage by holding rows scanned and fields scanned constant.

3.3.1 Vary Columns Scanned

The primary benefit of the column store design is its ability to read only the data for columns requested by a query. We show the effects of varying the number of scanned columns in Figure 2. For both graphs, we scanned 8 million rows.

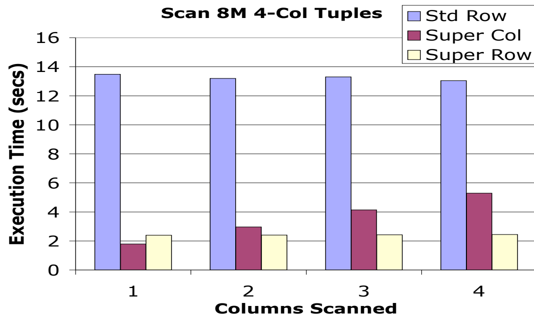
In Figure 2(a), we used 4-column tuples and varied the number of columns scanned. The standard row store takes more than twice the time of the super tuple row and column stores. When scanning one column, we see the column store is faster than the super tuple row store, but is slower for all other cases. Turning to Figure 2(b), we used a 32-column tuple and scanned 1, 8, 16, 24, and 32 columns. In this case, the column store enjoys a sizable performance edge over both the standard and super tuple row stores.

It is clear that C-Store performs extremely well when it scans a small fraction of the total number of columns in the table. This result puts us in a quandry as to how to show results for the remainder of the paper; scanning a small fraction of the columns will show the column store as relatively better performing for all cases, while scanning all columns will show the row store in a more favorable light. For our paper, we have opted to keep the optimizations separate and focus on performance trends for each storage choice individually. We therefore will scan all columns for each tuple width in all remaining graphs. The intent is to focus on the performance within each storage choice for a given storage optimization, rather than the relative performance of row and column stores.

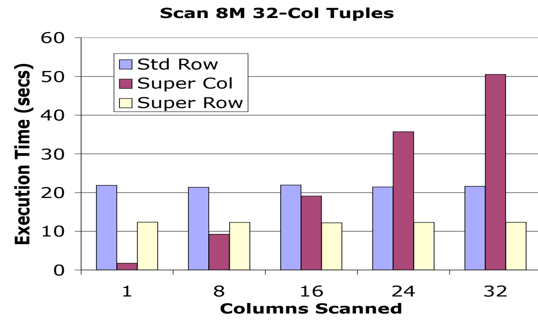
3.3.2 Constant Rows and Constant Fields

To demonstrate the benefit of using "super tuples" for a row store, we present two graphs in Figure 3. We varied the number of columns per tuple in both graphs, but held the number of rows constant in Figure 3(a) and the total number of fields constant in Figure 3(b).

When the number of rows is held constant, as in Figure 3(a), the amount of data being scanned doubles as the tuple width doubles. We see that the scan times for all storage choices are increasing as the tuple width increases. Interestingly, the standard row store takes 13 seconds to scan 8 million 4-column tuples, but only 21.6 seconds to scan 8 million 32-column tuples. Although we have increase the amount of data by a factor of eight, the scan time has not even doubled.

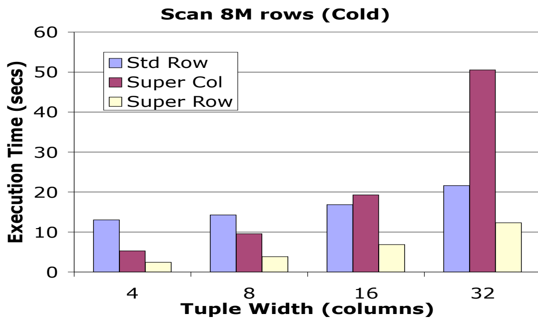


(a) 4-Column Tuples

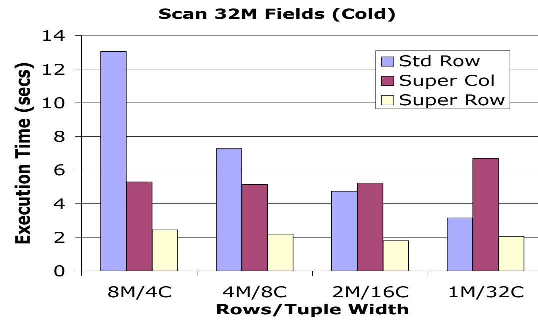


(b) 32-Column Tuples

Figure 2: **Execution times for varying number of columns scanned for an 8M row table without abstractions for (a) 4-Column and (b) 32-Column tuples.**

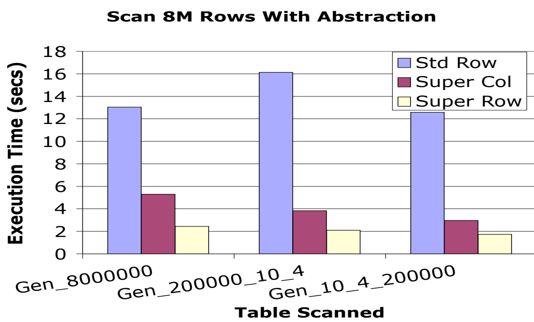


(a) Constant Row Cardinality

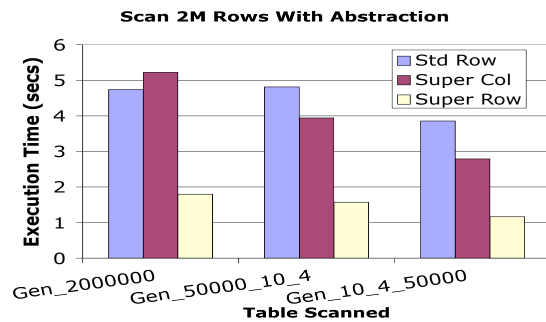


(b) Constant Data Size

Figure 3: **Super tuple effects when holding (a) rows and (b) fields constant. For the standard row store, small tuple sizes in both cases hurt performance.**



(a) 8M 4-Column Tuples



(b) 2M 16-Column Tuples

Figure 4: **Execution times for varying column abstractions for 32M fields using (a) 8M 4-Column and (b) 2M 16-Column tuples.**

Part of the reason the scan time does not increase as expected is that disk requirements are augmented by per-tuple overheads. Shore's per-tuple overhead is 16 bytes, which is 100% of our 4-column tuple and 25% of the 32-column tuple. Disk I/O costs alone are not enough to explain this behavior, however. We will revisit this issue later in the paper.

Figure 3(b) deals with varying the tuple width while the number of total fields remains constant. Holding the number of fields constant as the tuple width increases implies that the number of rows must decrease. We scanned 8 million 4-column tuples, but only 1 million 32-column tuples. We held the total number of fields ($rows * columns$) scanned constant at 32 million. Again we saw that the super tuple row store is the fastest in all cases. In fairness to the column store, these experiments were the worst case for that storage choice. We expected that scan times for all storage choices would stay roughly constant for a constant data size. While we saw constant scan times for the column store and the super tuple row store, the scan times for the standard row store dramatically decreased as the tuple width increased. Again, disk I/O is part of the story due to the elimination of 7 million per-tuple overheads. We saw a crossover point between the standard row store and the column store just below the 16-column tuple mark due to the marked decrease in standard row store scan costs.

For all of these experiments, we see that adding super tuples to standard row storage makes a significant difference in execution time for sequential scan.

3.4 Column Abstraction Effects

We now turn our attention to the effects of column abstraction. We generated synthetic data sets specifically to demonstrate how varying the amount of repeating data affects scan performance. We expect scan times to decrease as we increase the number of columns and the amount of data to be stored by using the column abstraction technique. To verify this hypothesis, we present two graphs in Figure 4. We hold the number of fields scanned constant at 32 million in both graphs.

Figure 4(a) shows three column abstraction choices for an 8 million row table with 4-column tuples. Gen_8000000 uses no abstraction to provide a baseline for comparison. Gen_200000_10_4 stores three abstraction levels with one column in the first level with 200000 unique values, one column in the second level with 10 values per first level tuple, and two columns in the leaf level with 4 values per second level tuple. This table is similar to the join cardinalities of Customer, Orders, and Lineitem from the TPC-H schema, respectively. Gen_10_4_200000 also has three abstraction levels, but has ten unique values at the first level, 4 second level tuples per first level, and 200000 leaf tuples per second level tuple. This table is more like the D4 view we used in Section 2 as an example, with L_RETURNFLAG at the first level, C_NATIONKEY at then second level, and L_EXTENDEDPRICE at the third level. As the amount of abstracted data increases, we see a general trend for the scan times of the column store and the super tuple row store to decrease. Interestingly, the scan time for the standard row store increases from Gen_8000000 to Gen_200000_10_4. We recall that column abstraction increases the total number of physical tuples for a row store. When combined with per-tuple storage overhead in the standard row store, it becomes clear why scan time might increase for certain data sets and abstraction levels.

The benefit of column abstraction with a standard row store depends on the number of additional tuples created by the process more than the savings in disk I/O. If disk I/O is the primary bottleneck, the standard row store should always be faster with column abstraction, not slower in some cases as seen in Figure 4. We break down the total scan time in Section 4 to identify the contributing factors.

4 Cost Model and Analysis

In Section 2, we presented the basic storage optimizations along with implementation-specific details for row and column stores. In Section 3, we identified several performance trends for the storage optimizations in isolation and combination. In this section, we develop a cost model for sequential scans for several reasons. First, it will verify our understanding of the costs that determine the relative performance of a standard row store and the super tuple row and column stores. Second, having an accurate cost model allows us to vary system parameters and/or properties of test data to forecast relative performance without actually building additional systems or loading the data.

At the most basic level, sequential scan is the most important factor in determining query performance. This is especially true when considering materialized views that have been created to exactly match the needs of a given query.

4.1 Cost Model Details

Our cost formulae depend on several variables, which we present in Table 4. The units for $SEQIO$, $RDMIO$, FC, and IC are "cost" units, which provide a basis for comparing scan costs relative to one another.

Figure 5 details the cost model for sequential scan of the traditional row store. We break each model down into three major contributing factors – disk I/O, iteration cost for the storage manager, and local per-tuple reconstruction cost. Tuple reconstruction, when necessary, consists of copying either a reference to the field value or the field value itself if it is small. We scale

Var	Description
$SEQIO$	Cost of a single sequential I/O
$RDMIO$	Cost of a single random I/O
$\|R\|$	Size of storage (pages)
$\|P\|$	Size of "super tuple" storage (pages)
$\|\|R\|\|$	Cardinality of table (tuples)
C	Width of row (columns)
F	Fraction of cold pages
S	Number of columns being retrieved
FC	Cost of function call
IC	Cost of storage manager iteration
n	Abstraction levels (1 means all cols in leaf)
$C(n)$	Columns in abstraction level n
$\|\|L(n)\|\|$	Average cardinality of abstraction level n (tuples)
$\|BP\|$	Size of buffer pool (pages)
$PGSZ$	Usable size of disk page (bytes)
CSZ	Column size (bytes)
OH	Tuple overhead (bytes)

Table 4: **Cost Model Variables**

$$\begin{aligned} SeqScan(StdRowStore) &= |R| * SEQIO * F && (1)\\ &+ \left(\sum_{i=1}^{n}\prod_{j=1}^{i}||L(j)||\right) * IC && (2)\\ &+ ||R|| * FC && (3) \end{aligned}$$

Figure 5: **Cost of sequential scan for standard relational storage with contributions from (1) Disk I/O, (2) Storage manager calls, and (3) Local per-tuple overhead**

disk I/O costs by the fraction of pages expected to be in the DBMS buffer pool already. At the extremes, $F = 1$ when all pages must be read from disk and $F = 0$ when all pages can be found in the buffer pool. A traditional row store must make a call to the storage manager layer for each row in the table. If the per-iteration overhead is high, these costs may even be significant when the buffer pool is cold.

Although column abstraction reduces or eliminates data duplication, the abstract columns must be stored. For example, if we are storing columns from Customers and Orders using column abstraction, we need to store a tuple for each Customer in addition to the tuple for each Order. However, using column abstraction may reduce the total number of disk pages ($|R|$), which will reduce disk I/O costs. With no column abstraction, we will have $n = 1$, $C(1) = C$ and $||L(1)|| = ||R||$, which simplifies the iteration cost to $||R|| * IC$.

We provide a cost model for the "super tuple" row store in Figure 6. We base disk I/O and storage manager calls on the number of packed pages. The improvement in storage manager calls is the primary benefit of the super tuple row store, especially for small tuples.

$$\begin{aligned} SeqScan(SuperRowStore) &= |P| * SEQIO * F && (4)\\ &+ |P| * IC && (5)\\ &+ ||R|| * FC && (6) \end{aligned}$$

Figure 6: **Cost of sequential scan for "super tuple" relational storage with contributions from (4) Disk I/O, (5) Storage manager calls, and (6) Local per-tuple overhead**

$$ABSAV = \sum_{i=1}^{n-1}\left(C(i) * CSZ * \left(||R|| - \prod_{j=1}^{i}||L(j)||\right)\right) \quad (13)$$

$$|R| = \frac{||R|| * (OH + C * CSZ) - ABSAV}{PGSZ} \quad (14)$$

$$|P| = \frac{||R|| * C * CSZ - ABSAV}{PGSZ} \quad (15)$$

Figure 8: **Calculations of (13) expected reduction in storage pages from abstraction, and resulting storage requirements for (14) regular and (15) "super tuple" storage.**

Finally, we provide the cost model for our "super tuple" column store in Figure 7. We make several assumptions in this cost model. First, we assume that disk storage is uniformly distributed among the columns, which is certainly not true when a column is run-length encoded. We also assume a uniform distribution of per-column contribution to the cost of local tuple reconstruction. Finally, we model prefetching of column data pages in accordance with our prototype implementation, as described in Section 3.1.

In Figure 8, we present a model for estimating the number of pages required to store a table based on the number of rows, columns, and average column size. These formulae could easily be inverted to estimate row cardinality based on a measured (or sampled) count of storage pages. $ABSAV$ is a calculation of the reduction in size given information about column abstraction. Note that the sum is from 1 to $n - 1$, so $ABSAV$ is zero without at least one level of column abstraction.

4.2 Model Validation and Prototype Performance Analysis

Our cost models attempt to capture performance trends as any set of variables change given constant values for the remaining variables. Before we begin the

$$|PS| = \frac{S}{C} * |P| \quad (7)$$

$$|PC| = \frac{|BP|/2}{S} \quad (8)$$

$$|RP| = \frac{|PS|}{|PC|} \quad (9)$$

$$SeqScan(SuperColumnStore) = (|RP| * RDMIO + (|PS| - |RP|) * SEQIO) * F \quad (10)$$

$$+ |PS| * IC \quad (11)$$

$$+ \frac{S}{C} * \sum_{i=1}^{n} \left(C(i) * \prod_{j=1}^{i} ||L(j)|| \right) * FC \quad (12)$$

Figure 7: **Cost of sequential scan for "super tuple" column storage with contributions from (7) Actual pages to scan, (8) Prefetch size per column, (9) Total random I/Os, (10) Disk I/O, (11) Storage manager calls, and (12) Local per-tuple overhead**

Var	Value
$SEQIO$	15000
$RDMIO$	450000
FC	6
IC	80
$\|BP\|$	16384 pages
$PGSZ$	32000 bytes
CSZ	4 bytes
OH	16 bytes

Table 5: **Prototype constant values for cost model variables**

validation of our models, we must determine constant values for our prototype. Table 5 shows the values we hold constant and the measured values we use for $SEQIO$, $RDMIO$, FC, and IC. Their relative values were calculated from measurements taken during a scan using the prototype system on our test hardware. The values would change given other hardware - for example, $SEQIO$ would increase relative to IC and FC if we had a single disk spindle instead of the large RAID-0 array.

In Figures 9 and 10, we show the predicted relative and actual prototype performance of scanning 4 and 16 columns, respectively, of the Gen_8000000 relation for our three page layouts. We see that the column store time increases as the number of column being scanned goes up. The increase is due mostly to the per-tuple local reconstruction cost. We also note that the cost of disk I/O decreases as the number of scanned columns decreases, as expected. Finally, we note the extremely high cost of tuple iteration for the standard row store. In contrast, tuple iteration is less than 1% of the total running time for both the column store and the "super tuple" row store. The cost model seems to track the three parts of total cost for both scans.

Figures 11 and 12 show the model prediction for scanning 32 million fields of data stored as 8 million 4-column rows and 1 million 32-column rows, respectively. In Figure 11 we again see the high iteration cost for the standard row store. In addition, the cost for disk I/O is very high for the standard row store compared to the "super tuple" column and row stores. Figure 12 tells a much different story. The model predicts that disk I/O is now roughly the same for each of the page layout choices. Iteration costs for the standard row store are much lower, while tuple reconstruction has increased for the column store.

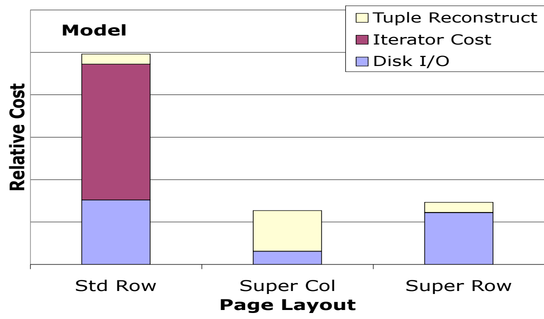
4.3 Model Forecasting

In Section 4.2, we validated our cost model against the Shore-based prototype system we created for experimental evaluation. In this section, we will change variables in the cost model to predict how systems with other characteristics would perform sequential scans.

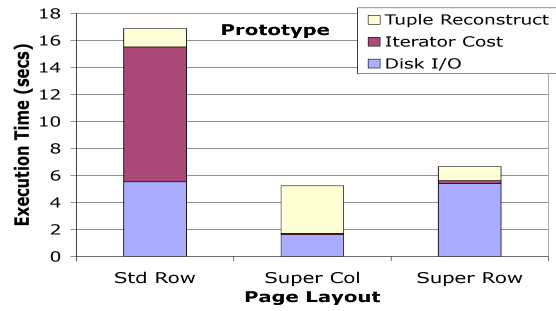
4.3.1 Sensitivity to Iteration Cost

Our experimental evaluation and cost model analysis demonstrates that using the Shore tuple iterator to scan a standard row store is CPU-bound. In fact, for our benchmark machine, iterating 1000 tuples on a page takes 5 times as long as reading the page from disk into the buffer pool! If possible, reducing per-tuple iteration cost for read-mostly workloads would provide a significant benefit even if no actual storage improvements are made.

Figure 13 shows the time for scanning 8 million 4-column tuples when the IC variable is 8 instead of the Shore value of 80. The cost model predicts that a sequential scan of all columns for the standard row store is now less than the column store scan time. Compare this graph to Figure 11(a) to see how dramatic the difference is. Reducing the iteration cost does not provide much performance improvement for the super tuple column and row stores – their iterations occur only once per disk page, not once per tuple. In fact, choosing the super tuple layout is a superior solution

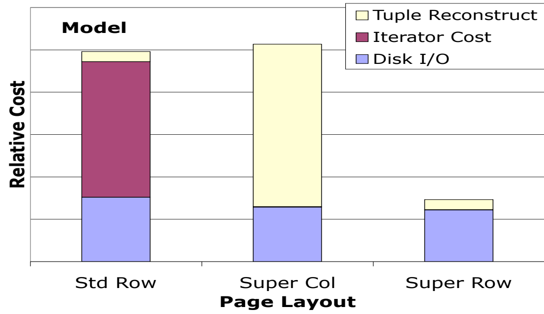


(a) Cost Model

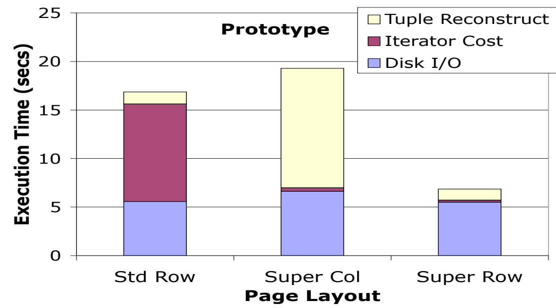


(b) Prototype

Figure 9: **Comparison of scanning an 8M row, 16 column table without abstractions scanning 4 columns using (a) Cost model and (b) Prototype.**

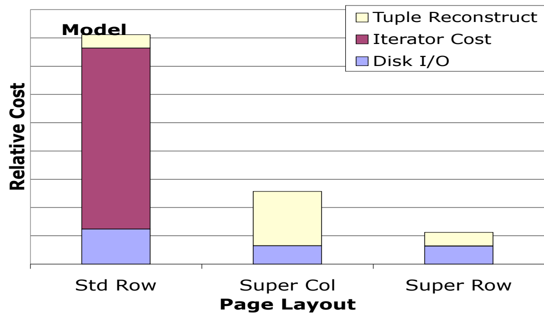


(a) Cost Model

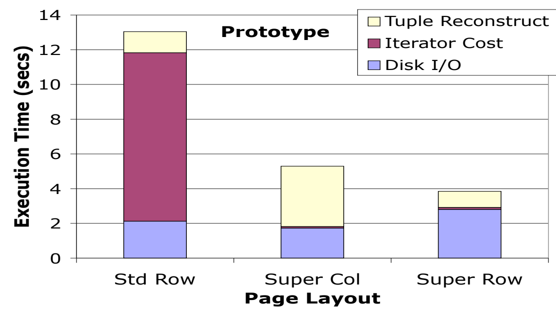


(b) Prototype

Figure 10: **Comparison of scanning an 8M row, 16 column table without abstractions scanning 16 columns using (a) Cost model and (b) Prototype.**

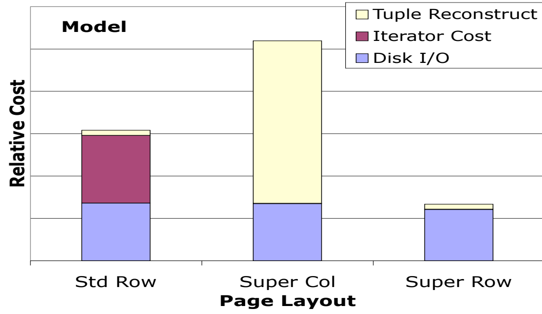


(a) Cost Model

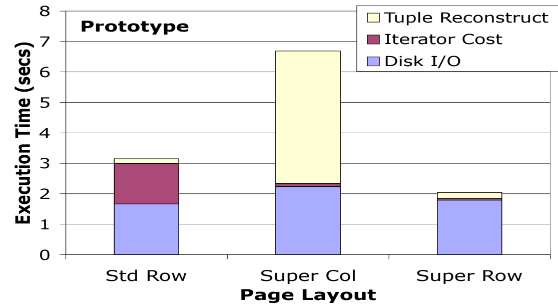


(b) Prototype

Figure 11: **Comparison of scanning all columns of an 8M row, 4 column table without abstractions using (a) Cost model and (b) Prototype.**



(a) Cost Model



(b) Prototype

Figure 12: **Comparison of scanning all columns of an 1M row, 32 column table without abstractions using (a) Cost model and (b) Prototype.**

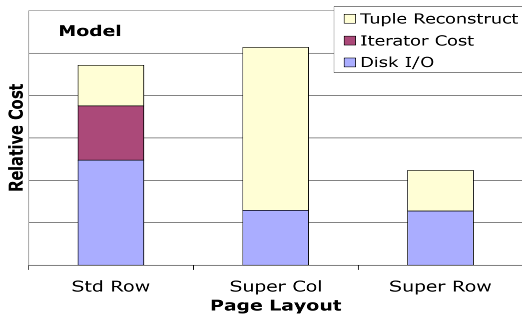


Figure 13: **Forecasted relative performance of scanning all columns of an 8M row, 4 column table without abstractions with $IC = 8$.**

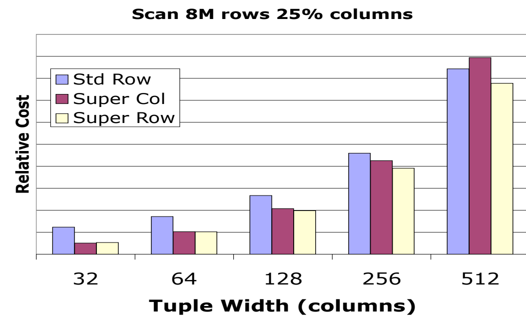


Figure 14: **Forecasted relative performance of scanning 25% of the columns of an 8M row table without abstractions as tuple width varies from 64 to 512 columns.**

to reducing per-row iterator costs, since the iteration cost is paid once per page regardless of the number of tuples on the page.

4.3.2 Sensitivity to Tuple Width

Our experiments vary tuple width from 4 to 32 columns. Using the model, we can forecast relative performance for the three storage formats for wider tuples. Figure 14 shows the cost model forecast for scanning 25% of the columns in 8 million tuples for tuple widths of 64, 128, 256, and 512 columns. Our model predicts that the overhead of tuple reconstruction for the column store increases until it is less expensive to scan using the standard row store with no improvements somewhere between 256 and 512 columns. As the tuple width increases, the number of tuples per page decreases and asymptotically approaches 1.

5 Related Work

Optimizing storage of one-to-many joins to avoid redundancy has been explored in the context of Non-First Normal Form databases. NFNF architectures allow nesting relations by permitting relation attributes to be defined as a set of tuples conforming to an arbitrary schema. In [6], Scholl et al. proposed a method for providing a logical relational view of data to the user while transparently storing a hierarchical clustering of related tuples as nested relations using a subset of the NFNF model for query optimization. Their proposal achieves a result similar to column abstraction and super tuples. However, their proposal is for base-table storage and not optimizing storage of materialized views. Further, their evaluation does not provide a direct comparison to an optimized column store system.

In [1], Ailamaki et al. evaluate CPU and cache-related overheads of various data page layouts, including row- and column-oriented choices. Their main contribution is a third choice called PAX, which combines the two by storing each column of a relation on a "minipage" within each physical disk page. PAX is effectively a column store within a row store. Choosing a PAX architecture would allow additional column encoding opportunities as it clusters values for each column domain together. The PAX concept is complementary to the row store optimizations presented in this paper, but its primary benefit is to increase

cache locality in the presence of query predicates. Our evaluation considers only sequential scan benefits to focus on materialized views designed to be scanned for matching and answering queries.

Fractured mirrors [5] store two copies of relations – one row-oriented and one column-oriented – to provide better query performance than either storage choice can provide independently. The mirroring also provides protection against data loss in the event of disk failure. The evaluation of the fractured mirrors work does not consider the column abstraction or super tuple optimizations of either the row or column stores.

The Bubba system [3] used a novel combination of inverted files and a "remainder" relation comprised of non-inverted attributes to store a relation. The inverted files are used as a data compression technique for attributes which contain redundant values. The inverted files are similar to a true column-oriented storage system, and capture the benefits of reducing disk I/O to improve sequential scan time. This work provides early motivation for the C-Store system for both column-at-a-time storage and data compression.

6 Conclusion

While prior work on column storage has clearly demonstrated the performance improvements it can deliver over row stores, the relative benefits of column stores and row stores have been obscured because there was no comparison in a common implementation framework. Further, several of the optimizations exploited by the C-Store proposal have analogues in row stores, but these row store optimizations were not considered. In this paper, we have attempted to shed light on the comparison between the two by implementing both in the same code base, and by defining and implementing the "super tuple" and "column abstraction" optimizations in the row store.

We noted several performance trends in our experimental evaluation. First, we verified the tremendous advantages of a column store system over a row store for workloads that access only a fraction of the columns of a table. Second, the "super tuple" optimization for the row store architecture appears to provide a significant performance benefit. Third, column abstraction can be used effectively to reduce storage needs for all storage choices, although its benefit is limited for a row store when used in isolation without super tuples. Finally, we showed that the contribution of CPU cost to total scan time can be a sizable component for scans of tables in a standard row store given a reasonably balanced hardware configuration with good sequential disk I/O performance, and that the super tuple optimization reduces CPU utilization in this case. We used our cost model to forecast the performance with a lightweight iterator and found that the row store architecture could be improved significantly without any changes to the underlying storage.

Many areas for future research are apparent. The crossovers in scan performance between super tuple-based row and column stores suggests that automatic storage selection for a given query workload would be beneficial for a system optimized for read-mostly query workloads. The cost model we devleoped in this paper can provide the basis for creating a storage selection "wizard." Note that selecting which views to materialize is an orthogonal issue – once the correct set of views is selected, one must still decide among the physical storage options.

We also note that column abstraction of one-to-many joins combined with super tuple-based row storage seems an ideal solution for efficient reconstruction of shredded XML documents or other complex entities. For normalized schemas which must frequently be re-joined but do not change frequently, choosing a super tuple-based materialized view as the primary storage for several tables in the schema may provide better performance.

References

[1] A. Ailamaki et al. Data page layouts for relational databases on deep memory hierarchies. *VLDB J.*, 11(3), 2002.

[2] M. J. Carey et al. Shoring up persistent applications. In R. T. Snodgrass and M. Winslett, editors, *SIGMOD Conference*. ACM Press, 1994.

[3] G. Copeland et al. Data placement in Bubba. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, Chicago, IL, June 1988. ACM Press.

[4] T. P. P. Council. TPC Benchmark H (Decision Support). http://www.tpc.org/tpch/default.asp, August 2003.

[5] R. Ramamurthy et al. A case for fractured mirrors. In *Proceedings of the 28th Internation Conference on Very Large Data Bases, Hong Kong, China; August 20-23, 2002.*

[6] M. H. Scholl et al. Supporting flat relations by a nested relational kernel. In *Proceedings of 13th International Conference on Very Large Data Bases, September 1-4, 1987, Brighton, England.*

[7] M. Stonebraker et al. C-store: A column-oriented dbms. In *Proceedings of the 31st International Conference on Very Large Data Bases, Trondheim, Norway, August 30 - September 2, 2005.*

[8] M. Stonebraker et al. C-Store System Source Code Version 0.1. http://db.csail.mit.edu/projects/cstore/, November 2005.