Column Stores - The solution to TB disk drives?

David J. DeWitt
Computer Sciences Dept.
University of Wisconsin
Problem Statement

- TB disks are coming!
- Superwide, frequently sparse tables are common
- DB performance is very sensitive to L2 cache performance
- For data warehouses, row stores are no longer the best way of mapping tuples to secondary storage
- Need techniques to make DB systems CPU bound
Over the last 30 years

DBMS architectures:

- Query engine
- Buffer pool

Hardware has changed dramatically

- CPUs: 1 MIP → 1 GIP
- Memory sizes: 2MB/CPU → 2GB/CPU
- Caches: 1K → 1MB
- Disks: 80 MB → 300 GB
- Network speeds: 1 Mbit/sec → 1 Gbit/sec
- Disk transfer rates: 1MB/sec → 60 MB/sec

NO significant changes (except for the use of parallel db techniques)
Assertion:

A radical redesign of DB system architectures is needed to deal with disk and CPU trends.

Conventional database systems (row stores) optimized for write intensive (e.g. OLTP) workloads and not read intensive workloads (e.g. Warehouse and CRM applications).

Column Stores are the most promising alternative out there.
Row Stores vs. Column Stores

Row Store - all attributes in a row (tuple) stored contiguously on disk

Column Store - all attributes of in a column stored contiguously on disk
Outline

- Technology trends working against Row Stores
- Transposed files - the birthplace of column stores
- From Transposed Files to C-Store
- Performance Results
Technology trends working against row stores

- “Supersized” disk drives
- Superwide, frequently sparse tables
- CPU/Memory Hierarchy trends
Disk trends over the last 25 years

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Capacity (MB)</td>
<td>81</td>
<td>330</td>
<td>73,400</td>
<td>300,000</td>
</tr>
<tr>
<td>RPM</td>
<td>3600</td>
<td>3600</td>
<td>10,000</td>
<td>10,000</td>
</tr>
<tr>
<td>Transfer rate (MB/sec)</td>
<td>1.209</td>
<td>1.25</td>
<td>30-58</td>
<td>39-80</td>
</tr>
<tr>
<td># of drives to hold a 100GB table</td>
<td>1235</td>
<td>304</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>Aggregate bandwidth (Mbytes/sec)</td>
<td>1493</td>
<td>380</td>
<td>90</td>
<td>60</td>
</tr>
<tr>
<td>Time to scan a 100GB table</td>
<td>1.1 min</td>
<td>4.3 min</td>
<td>18.5 min</td>
<td>27.8 min</td>
</tr>
</tbody>
</table>

- Effective disk speed is actually dropping
- Wide, sparse tuples make the situation worse
- Many queries touch only a small subset of the attributes
Factors Working Against Row Stores

- “Supersized” disk drives
- Superwide, frequently sparse tables
  - e.g. product catalog
- CPU/Memory hierarchy trends
Row Store Breakdown

- 10% Sequential Scan
- 10% 2ary index selection
- Join (no index)

- 6400 PII Xeon, NT 4.0, 8Kbyte pages
- Memory stalls are a huge performance hit!
L1 instruction and L2 data cache stalls dominate

Surprising differences between different systems

Why so bad????
Standard record layout for row stores

Employee Table

<table>
<thead>
<tr>
<th>RID</th>
<th>SSN</th>
<th>Name</th>
<th>Age</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1237</td>
<td>Jane</td>
<td>30</td>
</tr>
<tr>
<td>2</td>
<td>4322</td>
<td>John</td>
<td>45</td>
</tr>
<tr>
<td>3</td>
<td>1563</td>
<td>Jim</td>
<td>20</td>
</tr>
<tr>
<td>4</td>
<td>7658</td>
<td>Susan</td>
<td>52</td>
</tr>
<tr>
<td>5</td>
<td>2534</td>
<td>Leon</td>
<td>43</td>
</tr>
<tr>
<td>6</td>
<td>8791</td>
<td>Dan</td>
<td>37</td>
</tr>
</tbody>
</table>

Records are stored sequentially
Offsets to start of each record at end of page
Row Store Cache Behavior

select name from Emp
where age > 40

Result: one L2D fault per comparison
Summarizing

- Effective disk bandwidth when disk capacity is factored in is actually decreasing
- Wider tuples ==> fewer tuples per page ==> increased I/O cost
- Row stores have very bad L2 data cache behavior
Time line of relational storage alternatives

- **1968**
  - NSM (Row Store)

- **1971**
  - Transposed Files
  - (Lorie, IBM)

- **1985**
  - DSM
  - (Copeland & Hoshafian, MCC)

- **2000**
  - PAX
  - (Ailamaki, DeWitt & Hill)

- **2005**
  - Fractured Mirrors
  - (MIT, Brown, Brandeis)

- **2006**
  - C-Store
  - (Halverson, Beckmann, Naughton)

- **2003**
  - SuperRows + Column Abstraction
  - (Ramamuthy & DeWitt)
NSM, Transposed, & DSM Table Organizations

Each vertical partition is stored as a separate file.
DSM Implementation

Two copies of each column, one sorted on attribute value and one sorted on id value.

IDs need to "glue" columns back together again for NSM representation.

B-tree on ID could be sparse in which case IDs do not have to be stored in leaf entries.

<table>
<thead>
<tr>
<th>ID</th>
<th>A</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>A1</td>
</tr>
<tr>
<td>2</td>
<td>A2</td>
</tr>
<tr>
<td>3</td>
<td>A3</td>
</tr>
<tr>
<td>4</td>
<td>A4</td>
</tr>
<tr>
<td>5</td>
<td>A5</td>
</tr>
</tbody>
</table>

Logical View

Physical realization

Dense B-tree on A

Dense B-tree on ID
DSM Pros and Cons

**Pros:**
- Better I/O performance
  - Read only columns needed
- Better L2 data cache performance than NSM

**Cons:**
- Space overhead for IDs
- Cost of reassembling NSM tuples
- IDs hinder compression
- Not ideal for all query types.
  - Indexed selections with low selectivity factors
  - Scans involving all attributes

**Used by:**
- Bubba (MCC)
- Sybase IQ
- Monet DB X/100
DSM Reassembly Algorithms

- Naïve, attribute-at-a-time (standard approach)
- Chunk-based, k-way merge
- Scan of 1GB TPC-H Lineitem table, varying k, # of attributes returned
Apply concept of transposed files as the internal layout strategy for each NSM page.
PAX: Mapping to Cache

Fewer cache misses, low reconstruction cost

```
select name
from R
where age > 40
```
Effect on Accessing Cache Data

- PAX incurs 70% less data cache penalty than NSM
- PAX reduces cache misses at both L1 and L2
- Selectivity doesn’t matter for PAX data stalls
**Time and Sensitivity Analysis**

- **PAX**: 75% less memory penalty than NSM (10% of time)
- Execution times converge as number of attrs increases

---

**Execution time breakdown**

- **NSM**: Total clock cycles per record
- **PAX**: Total clock cycles per record

**Sensitivity to # of attributes**

- Elapsed time (sec)
- NSM and PAX curves show increase with number of attributes.

---

- Resource
- Branch Mispred.
- Memory
- Comp.
PAX: Summary

Pros

- Significantly improved L2D cache performance compared to both NSM and DSM
- More opportunities for compression than DSM since no IDs on each row
- Avoids extra I/Os of DSM to merge columns
- Faster than NSM for DSS queries
- Transparent to upper layers of DBMS

Cons

- Has same I/O performance as NSM (must read all columns)
Time line of relational storage alternatives

- **NSM (Row Store)**
  - **Transposed Files**
  - **DSM**
  - **PAX**
  - **Fractured Mirrors**
  - **C-Store**
  - **SuperRows + Column Abstraction**

- **1968**
- **1971**
  - (Lorie, IBM)
- **1985**
  - (Copeland & Hoshafian, MCC)
- **2000**
  - (Ailamaki, DeWitt & Hill)
- **2003**
  - (Ramamuthy & DeWitt)
- **2005**
  - (MIT, Brown, Brandeis)
- **2006**
  - (Halverson, Beckmann, Naughton)
Fractured mirrors

- Variant of RAID-0 with mirror being used for a DSM copy of each table

- Naïve implementation (load may not be balanced)
- Bubba hinted about this idea (but cannot find the reference)
- Use which ever copy is best for query being executed
Balanced Fractured Mirrors

- Balancing distributes random seeks
- Tuples partitioned across mirrors using round robin partitioning
Time line of relational storage alternatives

- NSM (Row Store)
- Transposed Files
- DSM
- Fractured Mirrors
- C-Store
- SuperRows + Column Abstraction

1968
1971 (Lorie, IBM)
1985 (Copeland & Hoshafian, MCC)
2000 (Ailamaki, DeWitt & Hill)
2003 (Ramamuthy & DeWitt)
2005 (MIT, Brown, Brandeis)
2006 (Halverson, Beckmann, Naughton)
C-Store/Vertica Features

- Hybrid, 2 level architecture
  - ROS - disk-resident column store based on transposed files
  - WOS - memory-resident row store
- Extensive use of compression to improve I/O performance
- Data replicated for performance and reliability
- Shared-nothing parallel architecture
- Targeted toward read-mostly environments
  - Warehouses
  - CRM Environments
C-Store Architecture

- **WOS** - write optimized store
  - Main-memory resident row store
  - Holds uncommitted and "recently" committed data
- **ROS** - read optimized store
  - Disk-resident column store implemented as transposed files (and not DSM)
- **Tuple Mover**
  - Periodically moves committed data from WOS to ROS
- **Shared Nothing Architecture**
- **Recovery via k-safety**
  - No logging
  - No 2phase commit
C-Store Data Models

Logical: relational tables

Physical:
- projections
- join indices (to glue projections together)
- B-tree indices on primary keys only
  - No secondary indices

Projection:
- Materialized view (from possibly multiple tables) stored as one transposed file per column in ROS
- Storage key for a row is its ordinal position (implied, not stored)
- Horizontally partitioned across multiple nodes
Examples

Given tables:  Emp (name, age, deptname, salary)
              Dept (deptname, floor)

Possible projections:
  Emp1 (name, age | age)
  Emp2 (name, deptname, dept.floor | dept.floor)
  Emp3 (name, salary | salary)
  Dept (deptname, floor | floor)

-Projections selected automatically using a workload-driven design wizard

-Join Indices/Permutations needed to “glue” projections together in some cases
### ROS compression schemes

<table>
<thead>
<tr>
<th>Sorted</th>
<th># distinct values</th>
<th>Few</th>
<th>Many</th>
</tr>
</thead>
<tbody>
<tr>
<td>Yes</td>
<td>Few</td>
<td>{ (val, pos, cnt) }</td>
<td>Delta encoded at block level</td>
</tr>
<tr>
<td>No</td>
<td>Few</td>
<td>{ (val, bitmap) }</td>
<td>NOT Encoded</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Bitmap compressed using RLE</td>
<td></td>
</tr>
</tbody>
</table>
## Compression Example

<table>
<thead>
<tr>
<th>Quarter</th>
<th>Product ID</th>
<th>Price</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q1</td>
<td>1</td>
<td>5</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>7</td>
</tr>
<tr>
<td>Q1</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>Q1</td>
<td>1</td>
<td>9</td>
</tr>
<tr>
<td>Q1</td>
<td>1</td>
<td>6</td>
</tr>
<tr>
<td>Q1</td>
<td>2</td>
<td>8</td>
</tr>
<tr>
<td>Q1</td>
<td>2</td>
<td>5</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>Q2</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>Q2</td>
<td>1</td>
<td>8</td>
</tr>
<tr>
<td>Q2</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Q2</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

- **Product ID**
  - (Q1, 1, 300)
  - (Q2, 301, 350)
  - (Q3, 651, 500)
  - (Q4, 1151, 600)

- **Price**
  - (1, 1, 5)
  - (2, 6, 2)
  - (1, 301, 3)
  - (2, 304, 1)

- **Price**
  - 5
  - 7
  - 2
  - 9
  - 6
  - 8
  - 5
  - ...
Other Techniques: Dictionary

<table>
<thead>
<tr>
<th>Quarter</th>
<th>Quarter</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q1</td>
<td>00</td>
</tr>
<tr>
<td>Q2</td>
<td>01</td>
</tr>
<tr>
<td>Q4</td>
<td>11</td>
</tr>
<tr>
<td>Q1</td>
<td>00</td>
</tr>
<tr>
<td>Q3</td>
<td>10</td>
</tr>
<tr>
<td>Q1</td>
<td>00</td>
</tr>
<tr>
<td>Q1</td>
<td>00</td>
</tr>
<tr>
<td>Q1</td>
<td>00</td>
</tr>
<tr>
<td>Q2</td>
<td>01</td>
</tr>
<tr>
<td>Q4</td>
<td>11</td>
</tr>
<tr>
<td>Q3</td>
<td>10</td>
</tr>
<tr>
<td>Q3</td>
<td>10</td>
</tr>
</tbody>
</table>

... + Dictionary

00: Q1
01: Q2
10: Q3
11: Q4
Operating Directly On Compressed Data Improves Performance

**Diagram Description:**
- The diagram compares the performance of different data compression techniques: No Compression, RLE Compression, Bit-vector compression, Dictionary single-value, and Dictionary multi-value.
- The x-axis represents the number of distinct values, ranging from 0 to 40.
- The y-axis represents the time in seconds, ranging from 0 to 20.
- Each compression technique is represented by a different line and marker:
  - No Compression: Solid black line
  - RLE Compression: Red line with triangles
  - Bit-vector compression: Blue line with stars
  - Dictionary single-value: Black line with crosses
  - Dictionary multi-value: Gray line with diamonds

**Query:**
```
SELECT colX, SUM(colX)
FROM tableX
GROUP BY colX
```
**Performance results**

<table>
<thead>
<tr>
<th></th>
<th>Oracle</th>
<th>Sybase IQ</th>
</tr>
</thead>
<tbody>
<tr>
<td>Constrained Space</td>
<td>164x</td>
<td>21x</td>
</tr>
<tr>
<td>Unconstrained Space</td>
<td>6.4x</td>
<td>16.5x</td>
</tr>
</tbody>
</table>

- Pretty much a “benchmark special”
- But, Alpha version of Vertica has better performance
- Apples-to-oranges comparison
- So, where does the performance boost come from?
An Apples-to-Apples Comparison

SIGMOD 06 submission by Halverson, Beckmann, and Naughton

Implementation of both column and row stores using SHORE storage manager

Three key factors at play:
- Use of SuperRows
- I/O savings due to use of transposed files
- Use of compression

Proposes a new storage layout for row stores termed a “Column Abstraction”
SuperRows

Observation: record overhead of standard slotted page format can be significant
- 8 bytes is probably typical (2 bytes of offset, 2 bytes of length, 4 byte tuple header)

Solution: SuperRows
- Array of fixed-length tuples stored as a single record on a slotted page
  - Variable-length tuples require a length field
- Can be used for both column stores and row stores
- PAX, Fractured Mirrors, & C-Store all use a form of this mechanism
- Saves overhead and iterator costs (as we will see)
Experimental Setup

- P4 2.4 Ghz Xeon, 1 GB memory, Raid-0 Volume using 6 250GB disk drives
- 32Kbyte SHORE pages
- 512 MB buffer pool
- 1/2 of buffer pool allocated for read-ahead
- Focus on sequential scans only
Scan 4 columns of a 8M row, 16 column table

Prototype

```
0 2 4 6 8 10 12 14 16 18
```

**Page Layout**

- Std Row
- Super Col
- Super Row

- Tuple Reconstruct
- Iterator Cost
- Disk I/O
Time line of relational storage alternatives

- **NSM (Row Store)**
- **Transposed Files**
- **DSM**
- **PAX**
- **Fractured Mirrors**
- **C-Store**
- **SuperRows + Column Abstraction**

- **1968** (Lorie, IBM)
- **1971** (Copeland & Hoshafian, MCC)
- **1985** (Ailamaki, DeWitt & Hill)
- **2000** (Ramamuthy & DeWitt)
- **2003** (MCC)
- **2005** (MIT, Brown, Brandeis)
- **2006** (Halverson, Beckmann, Naughton)
### Column Abstraction

#### Table View

<table>
<thead>
<tr>
<th>L-Returnflag</th>
<th>C-Nationkey</th>
<th>L-ExtendPrice</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>3</td>
<td>23</td>
</tr>
<tr>
<td>A</td>
<td>3</td>
<td>34</td>
</tr>
<tr>
<td>A</td>
<td>9</td>
<td>64</td>
</tr>
<tr>
<td>N</td>
<td>3</td>
<td>88</td>
</tr>
<tr>
<td>N</td>
<td>14</td>
<td>49</td>
</tr>
<tr>
<td>R</td>
<td>9</td>
<td>16</td>
</tr>
<tr>
<td>R</td>
<td>9</td>
<td>53</td>
</tr>
<tr>
<td>R</td>
<td>9</td>
<td>7</td>
</tr>
<tr>
<td>R</td>
<td>11</td>
<td>63</td>
</tr>
<tr>
<td>R</td>
<td>21</td>
<td>72</td>
</tr>
<tr>
<td>R</td>
<td>21</td>
<td>72</td>
</tr>
</tbody>
</table>

#### Physical Realization

<table>
<thead>
<tr>
<th>L-Returnflag</th>
<th>C-Nationkey</th>
<th>L-ExtendPrice</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>3</td>
<td>23</td>
</tr>
<tr>
<td>A</td>
<td>3</td>
<td>34</td>
</tr>
<tr>
<td>A</td>
<td>9</td>
<td>64</td>
</tr>
<tr>
<td>N</td>
<td>3</td>
<td>88</td>
</tr>
<tr>
<td>N</td>
<td>14</td>
<td>49</td>
</tr>
<tr>
<td>R</td>
<td>9</td>
<td>16</td>
</tr>
<tr>
<td>R</td>
<td>9</td>
<td>53</td>
</tr>
<tr>
<td>R</td>
<td>9</td>
<td>7</td>
</tr>
<tr>
<td>R</td>
<td>11</td>
<td>63</td>
</tr>
<tr>
<td>R</td>
<td>21</td>
<td>72</td>
</tr>
<tr>
<td>R</td>
<td>21</td>
<td>72</td>
</tr>
</tbody>
</table>

- Only shaded data actually gets stored
- Page level compression
- Scan interface reconstructs NSM record on the fly
- Similar idea in Peterlee relational engine (early 1970s)!!!!
Impact of Column Abstraction

Scan 8M Rows With Abstraction

Table Scanned

- Gen_8000000
- Gen_2000000_10_4
- Gen_10_4_200000
- Gen_10_4_200000

Chart showing the comparison of Std Row, Super Col, and Super Row for different table scanned.
Summary

- Disks keep getting bigger and slower
  - “Disks have become tapes” [Gray]
  - Column storage appears to be the best alternative
  - For read-intensive environments exploit huge disks to store multiple copies of the same data in different representations

- Surrogates (ie. rowids) in DSM are:
  - Not really necessary
  - Consume I/O bandwidth
  - Pollute the L2 data cache
  - Hurt compression

- Compress everything
  - CPU cycles are cheap
  - Disks are slow

- Operate on compressed data whenever possible
Some Interesting Open Problems

- Workload driven DB designer
- Materialization alternatives for column stores (early vs. late)
- C-Store compression techniques applied to PAX
- What does the DB designer selecting a particular projection imply about a pure column store vs. SuperRows with column abstraction?
- Other implementation strategies for column abstractions
- PAX vs. SuperColumns vs. SuperRows with abstraction
Acks

Many thanks to Daniel Abadi (MIT), Natassa Ailamaki (CMU), Alan Halverson (Wisconsin), Ravi Ramamurthy (MS), and for letting me “borrow” slides from some of their talks
Effect of Record Size
10% Sequential Scan

- **L2 data misses / record**
  - System A
  - System B
  - System C
  - System D

- **L1 instruction misses / record**
  - System A
  - System B
  - System C
  - System D

- **Notes**:
  - L2D increase due to locality + page crossing (except System D)
  - L1I increase due to page boundary crossing costs
**Disk trends over the last 25 years**

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Model</td>
<td>CDC 9760</td>
<td>Seagate 4384</td>
<td>IBM Ultrastar</td>
<td>Seagate Cheetah</td>
</tr>
<tr>
<td>Capacity (MB)</td>
<td>81</td>
<td>330</td>
<td>73,400</td>
<td>300,000</td>
</tr>
<tr>
<td>RPM</td>
<td>3600</td>
<td>3600</td>
<td>10,000</td>
<td>10,000</td>
</tr>
<tr>
<td>Transfer rate (MB/sec)</td>
<td>1.209</td>
<td>1.25</td>
<td>30-58</td>
<td>39-80</td>
</tr>
<tr>
<td>Normalized transfer rate</td>
<td>0.0149</td>
<td>0.0038</td>
<td>0.0006</td>
<td>0.0002</td>
</tr>
<tr>
<td>(Transfer Rate/Capacity)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Average Seek (ms)</td>
<td>30</td>
<td>14.5</td>
<td>4.9</td>
<td>5</td>
</tr>
<tr>
<td>Accesses/sec</td>
<td>33.33</td>
<td>68.97</td>
<td>204.08</td>
<td>200</td>
</tr>
<tr>
<td>Normalized access rate</td>
<td>0.412</td>
<td>0.209</td>
<td>0.003</td>
<td>0.0007</td>
</tr>
<tr>
<td>(Accesses per second/Capacity)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Effective disk speed is actually dropping
- Wide, sparse tuples make the situation worse
- Many queries touch only a small subset of the attributes
Role of Tuple Mover

- Operates as background batch process
- Periodically moves large batches of tuples from the WOS to the ROS
- Combines multiple ROS segments of a column into a single larger ROS segment
Other Techniques: Bit-vector

<table>
<thead>
<tr>
<th>Product ID</th>
<th>ID: 1</th>
<th>ID: 2</th>
<th>ID: 3</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
Scan 8M rows (Cold)

- Std Row
- Super Col
- Super Row

Tuple Width (columns)
Scan all columns of a 16 column table with 8M rows

<table>
<thead>
<tr>
<th></th>
<th>Std Row</th>
<th>Super Col</th>
<th>Super Row</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tuple Reconstruct</td>
<td>10</td>
<td>20</td>
<td>5</td>
</tr>
<tr>
<td>Iterator Cost</td>
<td>5</td>
<td>15</td>
<td>1</td>
</tr>
<tr>
<td>Disk I/O</td>
<td>5</td>
<td>5</td>
<td>3</td>
</tr>
</tbody>
</table>

**Page Layout**

![Graph showing Prototype costs for different page layouts]
Evaluation Using a DSS Benchmark

- 500M TPC-H DB
- Ran Queries:
  - Range Selections w/ variable parameters (RS)
  - TPC-H Q1 and Q6
    - sequential scans
    - lots of aggregates (sum, avg, count)
    - grouping/ordering of results
  - TPC-H Q12 and Q14
    - (Adaptive Hybrid) Hash Join
    - complex 'where' clause, conditional aggregates
- PII Xeon running Windows NT 4
- Used processor counters
Elapsed Execution Time

PAX improves performance up to 42% even with I/O
Column Abstraction

Consider view D4 from C-Store Paper:

Create View D4 as
Select L_Returnflag, C_Nationkey, L_ExtendPrice
From Customer, Orders, LineItem
Where C_CustId = O_CustId and
  O_OrderId = L_OrderId
Order by L_Returnflag, C_NationKey