Indexes Recap

	Heap File	B+Tree	Hash File
Insert	O(1)	O($\log_B$ n)	O(1)
Delete	O(P)	O($\log_B$ n)	O(1)
Scan	O(P)	O($\log_B$ n + R)	-- / O(P)
Lookup	O(P)	O($\log_B$ n)	O(1)

n : number of tuples
P : number of pages in file
B : branching factor of B-Tree
R : number of pages in range

Spatial/Multidimensional Indexes

Can index multi-dimensional data in a B-Tree, by creating composite keys (e.g., concatenate x/y coordinates together). But data will be ordered by X then Y (or Y then X), which will be in efficient for finding data in a narrow range of Y (or X).

Slides show 2 such indexes, one of which was assigned readings: R*Trees and QuadTrees.

R*Trees don't ensure non-overlapping bounding boxes; paper talks a lot about how to optimize construction of bounding boxes. QuadTrees just recursively subdivide space until some minimum density is reached.

Column Oriented Databases

So far we have studied physical data layout / heap files where data is laid out in a particular way: namely each row is consecutive in the heap file. But we could do something else -- and that's column stores. Let's understand the need for them by looking at the database market.

Roughly divides into two major use cases:

1. Transaction processing -- lookups of one or a few records, lots of small updates

2. Analytics ("warehousing") -- large scans of data, (essentially) append-only, often batch loads

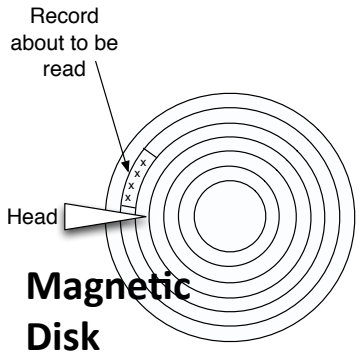
Together these represent about 90% of database sales, roughly evenly divided.

Let's think about scan performance:

Time is proportional to the number of records read.

Even if all columns aren't needed!

Why? Consider a magnetic disk -- entire record will pass under the head. Disk blocks are 512+Bytes, so that's the minimum size we can read.



Take a simple stock table:

tickstore(symbol,price,quantity,exchange,date,...)

And a query like

select avg(price) from tickstore where symbol = 'gm' and date = '1/17/2012'

Column store idea: store each column in a separate file
now query only needs to 3/5 of the data (assuming it is answered through a sequential scan.)

In reality large tables are often 100s of columns -- so this can easily save orders of magnitude in I/O on sequential scans.

When is this a good idea?

-- if updates of individual records are infrequent (since updates will need to touch multiple files now) -- batch updates are fine

-- if queries are mostly scan oriented

A bad idea w/ individual record lookups (since have to go to multiple pages), or when doing lots of small updates.

Column stores are a *perfect* fit for traditional warehousing applications.

Over the past 8 or so years have come to dominate this business; offer 10--100x better performance than the "row oriented" designs we have been studying on this kind of workload.

C-Store was a system designed to address all of these issues.

Key features included:

- Column-oriented query executor

- Compression aware query execution

- Write optimized storage system

- Shared nothing horizontal partitioning for distributed execution

- Ability to run read-only transactions without locking

- Automatic database designer

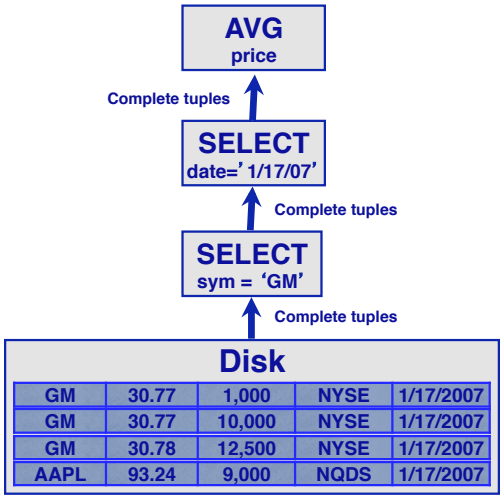
Since we haven't talked about distributed execution or transactions yet going to focus on the first three.

Executor:

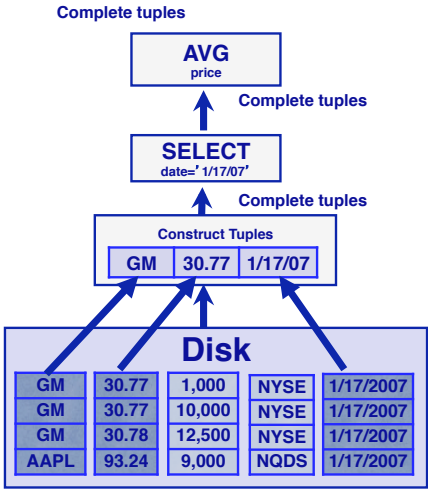
Consider the query

```
SELECT avg(price)
FROM tickstore
WHERE symbol = 'GM'
AND date = '1/17/2007'
```

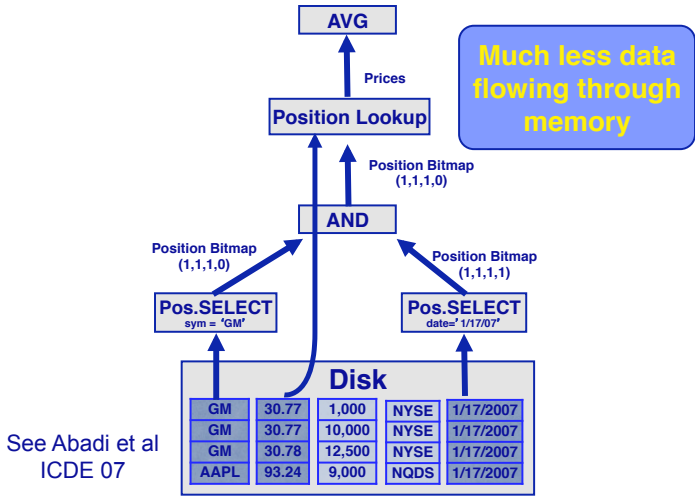
A traditional executor would do:



Simples way to build a column oriented design would be "early materialization"



In contrast, what C-Store does is "late materialization":



Reduces the amount of data flowing through memory all through the pipeline and also facilitates direct operation on compressed data:

<u>So how does compression work?</u>
Idea is that the query processor can keep data compressed, and operate directly on compressed data.

Multiple compression types:

- Run length encoding (1110000 --> 3x1,4x0)
- Gzip
- Delta Encoding (1.1, 1.2, 1.3 -> 1.1, +.1, +1)
- Block Dictionary ("sam", "sam", "joe" --> 1,1,2)
- Bitmap encoding ("M","M","F"F" --> 1100,0011)
- …

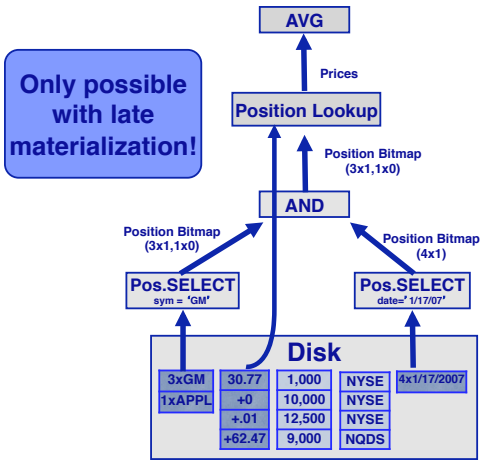
Idea is that each column can be compressed using the data that is best for it. So, for example, if data is few valued and sorted, RLE will work great. If data is many valued and sorted, delta may be better. For unsorted data, Gzip may be best.

Obviously sorted data compresses MUCH better. Note that we can store sorted data because we don't expect a lot of updates or inserts (will return to how we handle loading in a minute).

Note that we can "secondarily sort" data, where we sort on col A then col B, which will result in B being "partially sorted". E.g., " A 1, A 2, B 1, B 2", and we can still get advantages of compression on these secondary sort columns.

Really cool thing is that we can operate directly on compressed RLE and dictionary data.

For example, can keep position bitmaps RLE encoded in previous example.



Other examples:

- Can do position lookup on bitmap and dictionary data

- Direct aggregation on RLE and dictionary data

- Join runs of compressed RLE and dictionary data

- Min/max extracted directly from sorted data

Compression and sorting can be a huge win (show slide)

So how do we increase the amount of sorted data?

Store duplicate copies in different physical orders

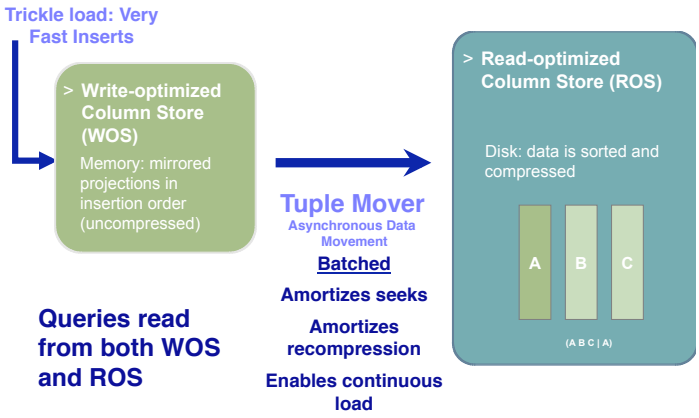
Don't need to store all columns in these replicas -- can create "projections" that are subsets of the columns. **Database designer** tries to do choose the sort orders and projects that are optimal for a historical workload.

We can use these replicas both for performance and recovery!

Ok -- so now we've seen how query execution and performance are integrated. Let's talk a little about writes.

Problem is that if system is compressed and sorted, inserts will be very expensive. Even if we batch load, we don't want to rewrite everything whenever data is added.

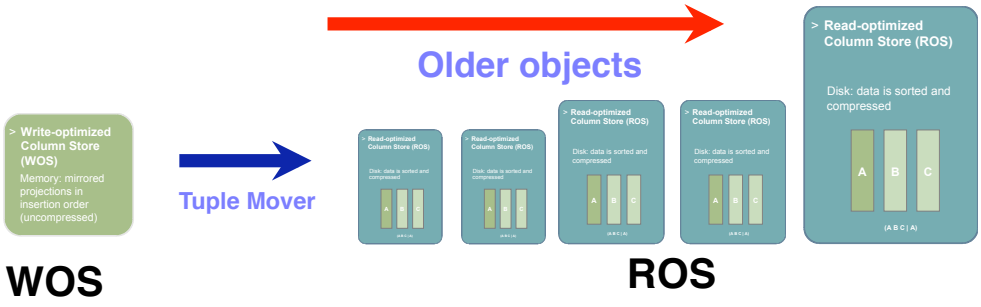
So the idea is that there is a "write optimized" store sitting in front of the main "read optimized" database.



But this doesn't quite solve all of the problems, because we still seem to have to rewrite whole WOS when tuple mover runs. So what to do?

Idea: store multiple ROS objects, instead of just one. Scan each one to answer a query. Tuple mover can now create new ROS objects, rather than merging into main object.

Must periodically merge ROS object to limit the number of objects that must be scanned.



Now let's turn to the paper that was assigned this time -- "**Column-Stores vs. Row-Stores: How Different Are They Really**?"

What is the key idea?

To study whether its possible to emulate the performance of column stores in existing row oriented databases.

How did we do this?

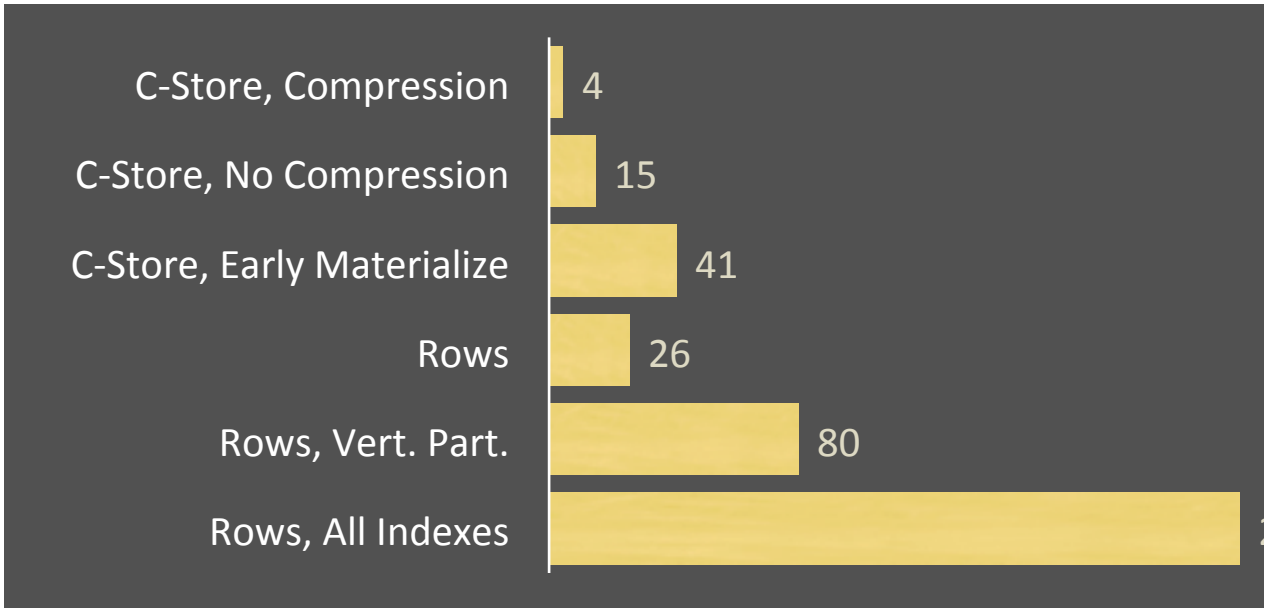
Hired a professional DBA to help try out different physical database designs in Oracle.

What are the different ways you might emulate a column store?

Index only plans -- create an index for every column, and retrieve values from leaves of index

Vertical partitioning -- create a one column table for each column

How do these systems stack up? (Show slide)



What's going on here?

Vertical partitioning does badly because of two factors:

1) Tuple headers
Total table is 4GB
Each column table is ~1.0 GB
Factor of 4 overhead from tuple headers and tuple-ids

2) Merge joins
Answering queries requires joins
Row-store doesn't know that column-tables are sorted
Sort hurts performance

Would need to fix these, plus add direct operation on compressed data, to approach C-Store performance

Index only plans do worse!

Consider the query:

SELECT store_name, SUM(revenue) FROM Facts, Stores
WHERE fact.store_id = stores.store_id AND stores.country = “Canada”
GROUP BY store_name

Two WHERE clauses result in a list of tuple IDs that pass all predicates
Need to go pick up values from store_name and revenue columns
But indexes map from value-->tuple ID!

Column stores can efficiently go from tuple ID-->value in each column

Could we get C-Store like performance in a Row Store?

Maybe.

- Need to store tuple headers elsewhere (not require that they be read from disk w/ tuples)
- Need to provide efficient merge join implementation that understands sorted columns
- Need to support direct operation on compressed data
--- > Requires “late materialization” design