Some Reminders for a Seamless Online Class…

- Please turn on your video
- Mute yourself (press and hold spacebar when you'd like to talk)
- Don't do anything you wouldn't do in an in-person class
- I will occasionally check the chat for messages if you'd like to share there instead
- Please say your name before you speak

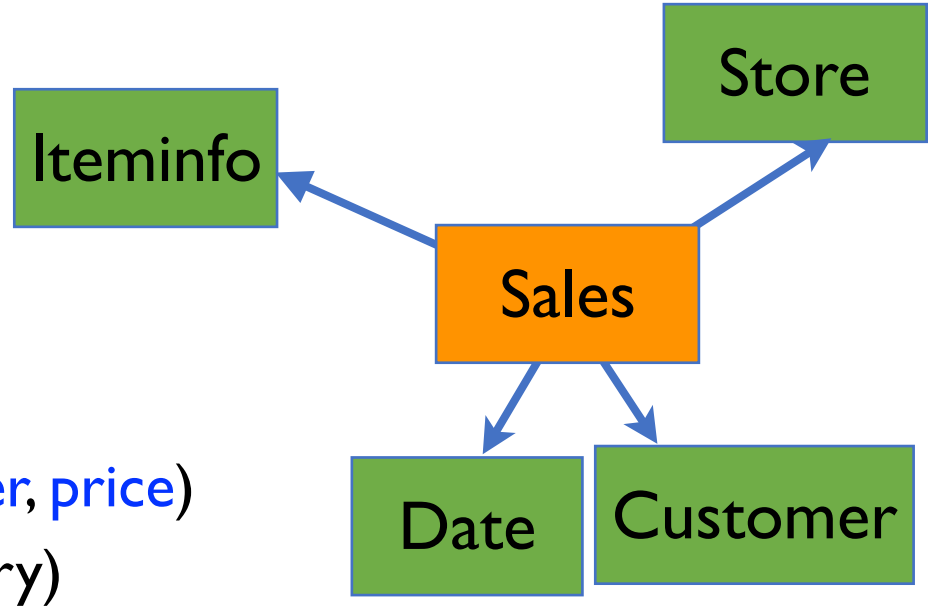
Recap

- Data-savviness is the future!
- "Classical" relational databases
  - Notion of a DBMS
  - The relational data model and algebra: bags and sets
  - SQL Queries, Modifications, DDL
  - Database Design
  - Views, constraints, triggers, and indexes
  - Query processing & optimization
  - Transactions
- Non-classical data systems
  - Semi-structured data and document stores
  - Unstructured data and search engines
  - Cell-structured data and spreadsheets
  - Dataframes and dataframe systems
  - OLAP, summarization, and visual analytics
  - **Compression and column stores**

Remembering OLAP

- What is OLAP?
  - A specialization of relational databases that prioritizes the reading and summarizing large volumes (TB, PB) of relational data to understand high-level trends and patterns
    - E.g., the total sales figures of each type of Honda car over time for each county
  - "Read-only" queries
- Contrast this to OLTP: OnLine Transaction Processing
  - "Read-write" queries
  - Usually touch a small amount of data, e.g., append a new car sale into the sales table

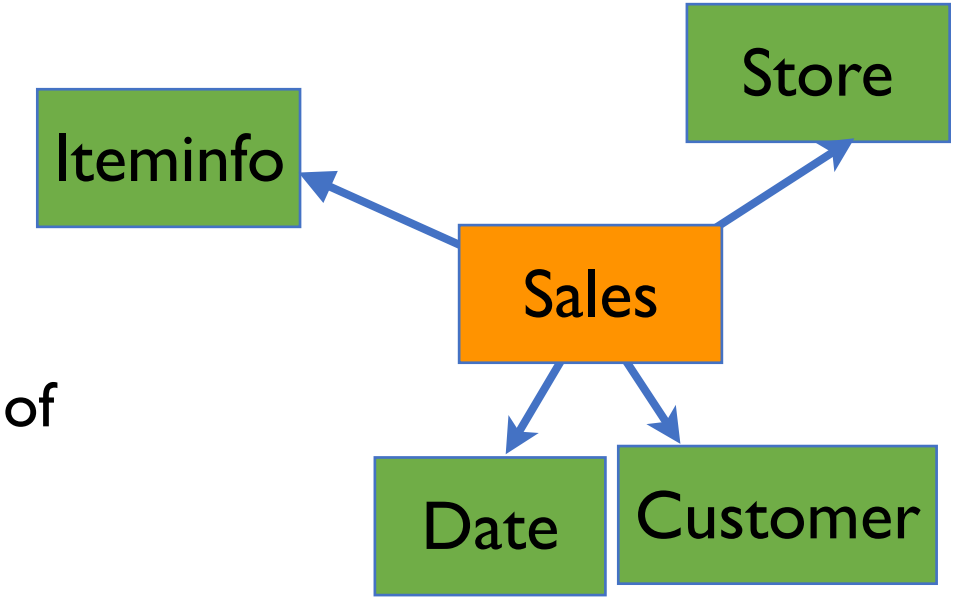
Star Schema



- Let's take a canonical example for a data warehouse:
  - Fact table: Sales (itemid, storeid, customerid, date, number, price)
  - Dim table: Iteminfo (itemid, itemname, color, size, category)
  - Dim table: Store (storeid, city, state, country)
  - Dim table: Customer (customerid, name, street, city, state)
  - Dim table: Dateinfo (date, month, quarter, year)

- Reminder: a fact table talks about information about an event
  - Dimension table records auxiliary information about an event
- Even this simple example has lots of attributes.
- Most fact and dimension tables in practice can have 100s of attributes each

Star Schema



- Most fact and dimension tables in practice can have 100s of attributes each
  - For now, let's just focus on the fact table

- However, most analytical queries don't require accessing all 100-odd attributes across all tables
- Rarely will you be doing SELECT * on a 1PB Sales fact table.
- Instead, most analytical queries would touch 2-3 attributes at a time.
  - SELECT category, country, month, COUNT(number) FROM Sales NATURAL JOIN Iteminfo NATURAL JOIN Store GROUP BY category, country, month

Traditional Storage: Row-oriented

- For now, let's focus on a 100+attribute fact table.

- Traditional relational databases store data in a row-oriented manner across pages on disk
- Typical page:
  - <Metadata (# of tuples, pointer to start of each tuple><All attributes of tuple 1><All attributes of tuple 2>… in some order
  - Often separating out variable length and constant length fields

- So, a GROUP BY query touching only 3 attributes will still read the remaining 97 attributes
- These remaining attributes will get projected out at some point.
- Two reasons for unnecessary work:
  - Reading redundant data from disk
  - Having to throw away this data within memory

Column-Oriented Storage

- Storing columns separately from each other
- For example, if the only query I issue to the fact table of 100+ attributes is:
  - SELECT itemname, color, SUM(number) FROM Sales GROUP BY itemname, color

- Then, storing itemname, color, and number separate from the rest, will help me avoid reading and processing 97% of the underlying data
  - <Metadata (# of tuples, pointer to start of each tuple><3 attributes of tuple 1><3 attributes of tuple 2>… ==> *only need to read this*
  - <Metadata (# of tuples, pointer to start of each tuple><97 other attributes of tuple 1><97 other attributes of tuple 2>…
  - …

Column Stores

- Column stores are a specialization of relational databases for OLAP that use column-oriented storage (among many other innovations)

- Many industrial offerings: Vertica, Vector, Druid, Greenplum, Amazon Redshift, SAP IQ, ParAccel …
- Column store ideas have made their way into traditional relational databases like PostgreSQL, MS SQL Server, DB2, Oracle, …
- Column storage ideas have been used in non-relational settings as well, e.g., Dremel, Impala, HBase, …
- Ideas have been around for >2 decades but really rose to prominence in the late 2000s

Column-Oriented Storage

- OK, if we store columns separately from each other, how do we reconstruct the tuples?

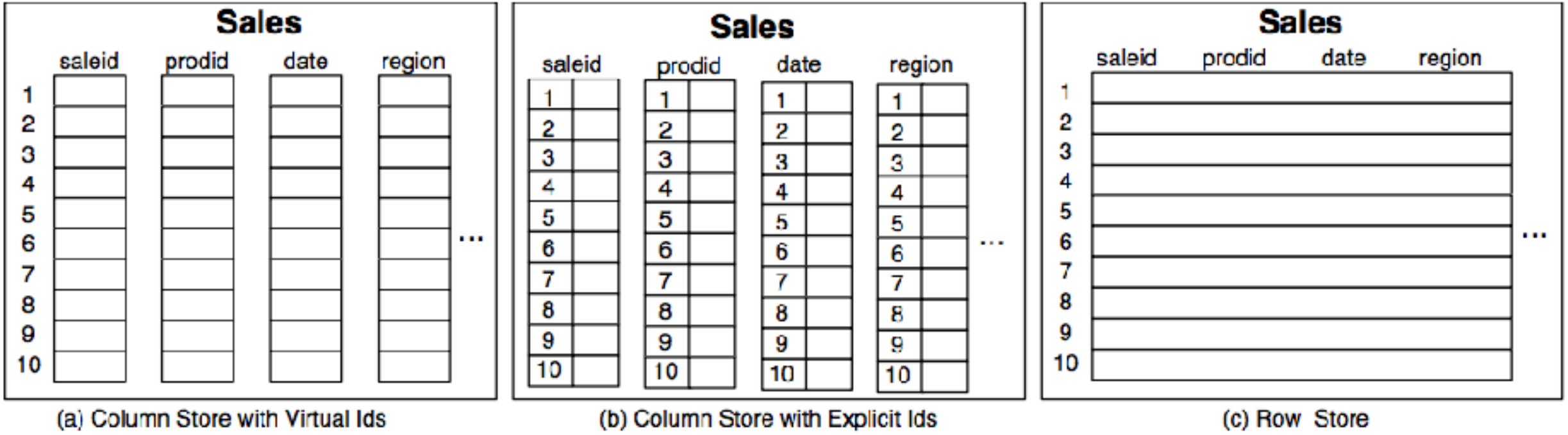
- One option, explicit IDs.

(b) Column Store with Explicit Ids

(c) Row Store

From the book "The Design and Implementation of Modern Column-Oriented Database Systems"

Another Option: Virtual IDs

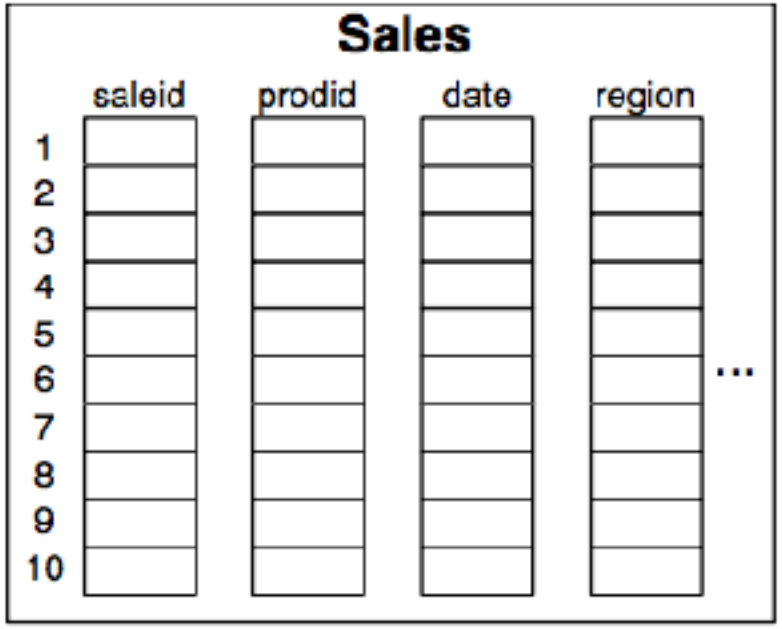


(a) Column Store with Virtual Ids

(b) Column Store with Explicit Ids

(c) Row Store

Virtual IDs



(a) Column Store with Virtual Ids

- Downside of actually storing a tuple ID is the extra storage and processing overhead
- Instead, most column stores implicitly use position to record where the *n*th value for a given column.
  - If using fixed width values, this is easy to lookup
    - From the first position, do *n* x width.
  - Else, will need a bit more bookkeeping when executing queries across multiple columns (groups)

What are the drawbacks?

- If we simply organize the columns such that the columns often accessed together are stored together, do we have a better design than vanilla row stores?
  - What are ways this is actually worse than row stores?

What are the drawbacks?

- If we simply organize the columns such that the columns often accessed together are stored together, do we have a better design than vanilla row stores?
  - What are ways this is actually worse than row stores?
  - If you want to access a single tuple, in a row store, can jump to specific location and read all the relevant attributes
    - In column stores you'll need to do multiple random jumps to all the pertinent columns of interest
    - Specifically, hard to do updates: will need to individually update all of the column values
  - Better when we want to read a *subset of columns of most of the tuples*
  - Column stores are used primarily with analytical queries: queries that scan a large fraction of individual tables and compute aggregates or statistics across them
  - So let's focus on read-only OLAP queries for now…

OK, so we have separated out the columns. Are we done?

- Not quite.
- There are lots of additional improvements beyond columnar storage in column stores…

- First up, compression

Compression

- Q: How would you go about compressing a collection of salaries? states? product codes? zipcodes?

- Desirable properties for compression:
  - Must reduce space considerably
  - Must be quick to decompress
    - Most "fancy" approaches for compression don't help for interactive data processing

First Approach: Run-Length Encoding

- This approach simply encodes a sequence of values based on the number of times each value appears
  - A A A B B B B B A A A C
  - <3, A> <5, B> <3, A> <1, C>

- Q: When would this work well?
- Works really well when data is already pre-sorted by the attribute; doesn't work too well with randomized orders
- Doesn't work too well with numeric data with many unique values

- Advantage: can use it effectively if we want to do aggregates:
  - e.g., SUM(<3, 10> <5, 6>) = 3 * 10 + 5 * 6

Second Approach: Dictionary Encoding

- Mapping every distinct value into another value that takes less space.
- For example, instead of using variable length strings for encoding states, can encode states into a single byte (2^8 > 50)
  - Alabama = 0, Arkansas = 1, …
- Q: When does this work well?
- Strings where once again # of distinct values is small; doesn't require ordering

Third Approach: Frame of Reference

- Store first value, rest stored as "deltas" from the previous one
- e.g., 1000, 1001, 1003, 1004, …
  - Stored as: 1000, +1, +2, +1 etc.

- Q: When does this work well?
- When the data is already sorted, or if there is locality: works better with numbers than with strings

Fourth Approach: Bit Vector Encoding

- Encode every single possible value as a bit vector
- 1 1 3 2 2 3 1 encoded as:
  - Bit string for 1: 1 1 0 0 0 0 1
  - Bit string for 2: 0 0 0 1 1 0 1
  - Bit string for 3: 0 0 1 0 0 1 0
- Q: When does this work well?
  - Small number of distinct values
    - Or, more distinct values, but we will need to compress the bit strings themselves
- Benefit of the bit strings is that they can be easily processed on.
  - If we wanted to get all the values corresponding to 2, we simply need to get the bit string for 2
  - If we wanted 2 or 3, we can OR the two bit strings
- Bit vectors are very efficiently manipulated by modern processors

So: what compression should we use?

- Answer: it depends
  - On the type of data
  - The locality
  - What you want to do with it
  - The # of space you have
  - etc.

Now, back to Column Stores

- Since columns are stored separate from each other, we have more possibilities for compression than if we were to apply it to entire rows
- e.g., <Washington, M, 100000>, <California, F, 200000>, <Louisiana, O, 150000>
- Can't easily apply any of the aforementioned compression schemes at the row level
- But:
  - States (Washington, California, …) can be dictionary encoded
  - Gender (M, F, O, …) can be bit vector encoded
  - Salaries can be FOR-encoded
- This is why compression is more beneficial for column stores than row stores

Should we store entire columns separate from each other?

- Answer: it depends
  - Entire columns lead to more compression
  - But if multiple columns are often accessed together you might want to store them together — known in the literature as "column groups"

- In general, this depends on the workload.
  - What if you only issue two queries on a relation with attributes A—Z, one with A, B, C and another with A, B, D?
    - Can store <A, B, C, D> and then <E—Z>, or
    - Can store <A, B, C>, <D>, <E—Z>, (+symmetric alternative) or
    - Can store <A, B>, <C>, <D>, <E—Z>, or
    - Can store <A>, <B>, <C>, <D>, <E—Z>
  - The best choice depends on the frequency of the queries and the distributions of values in A, B, C, D, as well as combinations of A, B, C, D.

Column Stores

- Two key ideas so far:
  - Store (groups of) columns separate from each other
  - Columns can be compressed
- Other ideas…

Idea 1: Many Different Layouts

- The same column can appear multiple times.
- What if you only issue two queries on a relation with attributes A—Z, one with A, B, C and another with A, B, D?
    - New alternative: can store <A, B, C>, <A, B, D> and then <E—Z>

- Pushing this one step further, different "column groups" don't need to be stored in the same order
    - Can use order most beneficial for workload and for compression
    - But! There must be at least one set of column groups that allows us to reconstruct the overall relation
    - For example, can't store <A, B, C, ordered on A>, <A, B, D ordered on B>, <E—Z ordered on A>
        - If we ever get a query on A, B, D, E, we can't answer it because we can't correlate the A, B, D tuples with the <E—Z> tuples
        - Plus, we can't make updates to specific tuples
        - Remember: reconstructing relations after normalization!

Idea 2: Late Materialization (hard!)

- So far, we have assumed that column stores bring all the necessary attributes into memory and "reconstruct" a (projected-out) tuple and proceed as in regular row stores
- We don't need to do so in column stores; in fact, there are benefits to delaying materialization until later

- If we have a predicate on an attribute, we can process that attribute first, and then only append other attributes as needed for the tuples for which the predicate is satisfied

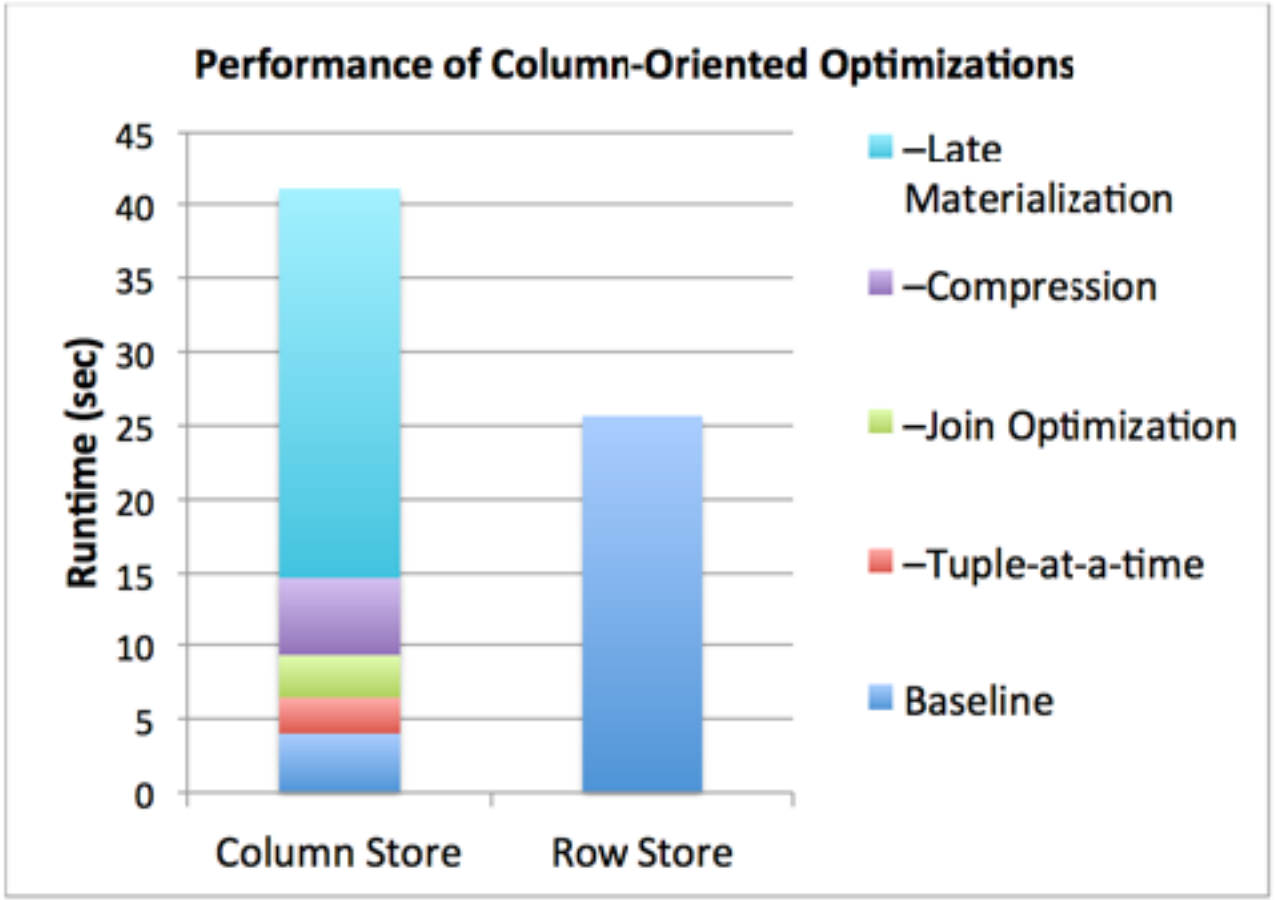
Idea 2b and 2c: Vectorized & Join Processing (hard!)

- Modern CPUs can do efficient parallel processing of multiple data items at once:
  - This is more efficient when you apply it on a column at a time rather than multiple columns
    - E.g., applying a predicate across a column
    - Otherwise you'll be skipping over "junk"
  - This further justifies late materialization
- Moreover, we can do more efficient join processing if we delay materialization:
  - "ship" the key values that need to be joined, and only materialize on demand

Experiment from Abadi et al. SIGMOD'08

- 60 M rows; SSBM benchmark, which uses narrow tables
- Performance as optimizations are "dropped"

- So benefits are not pronounced by simply doing columnar storage for this case
- Only appears when other optimizations are added



Idea 3: Updates via a WOS

- Many column stores also support inserts, deletes, updates
- For example in Vertica
  - Handled via a separate row-oriented portion called the Write-Optimized Store where all the new or updated tuples are stored
  - For "invalidating" existing records, we can simply maintain a vector of all row numbers that have been deleted/updated
    - So that they can suitably ignore those results during processing
  - This WOS data is periodically merged back into the columns

Takeaways

- For OLAP, column-oriented storage trumps row-oriented storage
  - Many tricks beyond splitting columns up
    - compression, late materialization, redundant layouts, vectorization, join processing, efficient write processing
- For OLTP, the costs of many random accesses for updating columns makes a columnar layout not worth it
  - Hence OLTP systems look more traditional, and typically opt for row-oriented storage
- Many systems are now opting for hybrid layouts to try to support both OLAP and OLTP in the same system

A Meta Question to Consider for OLAP

- We have now seen two approaches to supporting OLAP more efficiently:
  - First, materializing appropriate data cubes in advance for report generation, while making sure that those cubes can answer aggregate requests at various levels of the hierarchy that may be of interest
  - Second, columnar storage (plus other tricks) via column stores

- Which approach should we use?
  - A: it depends.
  - If we have a fixed set of queries (e.g., a dashboard that needs to be updated daily), materialization is a good option
  - If we have dynamically varying queries that each touch subsets of data, then a full-fledged columnar storage is a good option