Secondary Storage

Holger Pirk

Context

Secondary Storage is about replicating data

- The opposite of normalization
  - But in a controlled manner
  - The DBMS is in charge of replicas
  - They can be created and destroyed without breaking the system
  - They are semantically invisible to the user
  - They can be enormously beneficial for performance

However,

- They occupy space
- *They need to be maintained under updates*
- They stress the query optimizer
- They can only be used for certain operations

Purpose of this lecture

By the end, you should

- Understand the concept of secondary storage
  - *Indices*
  - Have heard of *Materialized Views*
- Be able to select the right secondary storage structure for a query/workload
- Understand why secondary storage is not always a good idea

Some Nomenclature

Definition

Clustered or Primary Index An index that is used to store the tuples of a table

- You can have (at max) one of these

Unclustered or Secondary Index An index that is used to store pointers to the tuples of a table

Context

- This class is about unclustered indices, all of the presented techniques can be used as clustered indices
- Focus is on concepts and data structures, not the SQL to create them

Maintaining indices in SQL

Creating them

```
CREATE INDEX index_name ON table_name (column1, column2, ...);
```

Dropping them

```
DROP INDEX index_name;
```

This isn't particularly useful

- Unclear what kind of index is created
- No control over parameters
- Virtually all systems provide much finer control (look at the documentation)

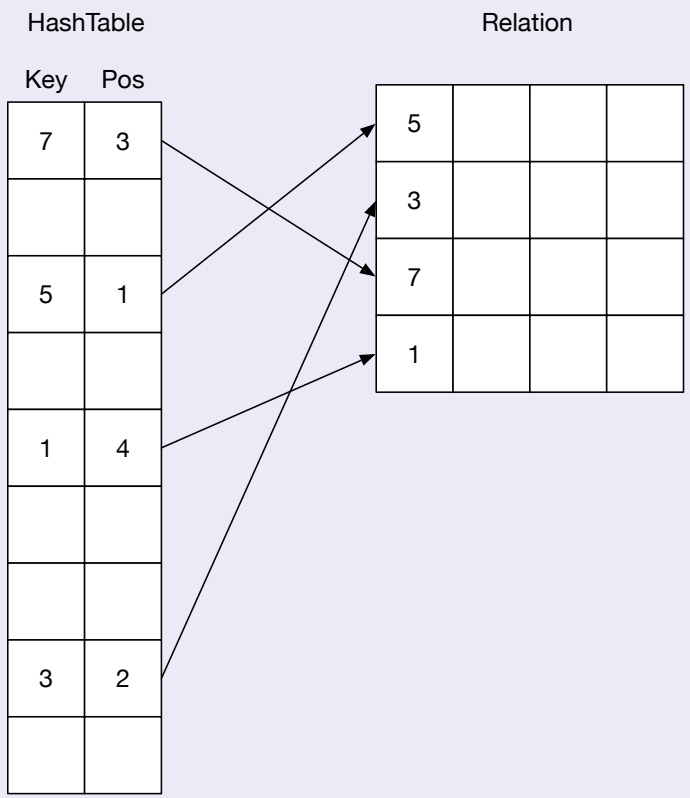
Hash-Indexing

Hash-Indexing

Remember Hash-joins?

- Step one was building a hash-table
- A hash-index is the same thing but persistent

Visualisation



Ephemeral hash-tables

For hash-joins, we were assuming a "stop-the-world" implementation

- there are no new tuples added during query evaluation
  - We knew (roughly) how many tuples are going to end up in the table
- The hash-table was discarded after the join
- we did not have to worry about updating it
- *If the hash-table is persistent, all of that changes*

Persistent hash-tables

The hashtable may grow arbitrarily large

- Overallocate by a lot
- If fill-factor grows beyond x percent (e.g., 50 percent), rebuild
  - Rebuilds can be very expensive (even with *consistent hashing*)
  - This leads to nasty load spikes

Tuples may need to be deleted

- Remember: we used empty slots as markers for the end of probe-chains
- On delete, we have to put a value in the slot
  - A forward marker, the *next* or the *last* value in the probe chain

Hash-Indices

Usefulness

- Remember: we said, hashjoins are good for equi-joins
  - Because hash-tables allow the quick lookup of a specific key
- Not useful for inequality-joins
  - Because hash-tables do not allow to find the, w.l.o.g., next greatest value
- The same applies here:
  - Persistent Hash-tables are great for hash-joins and aggregations (duh!)
  - The also help a lot to evaluate equality conditions:
    - `select * from customer where name = "holger"`
  - Not great for anything else
    - `select * from customer where id between 5 and 8`

Foreign-Key Indices

Foreign-Key Indices

In SQL

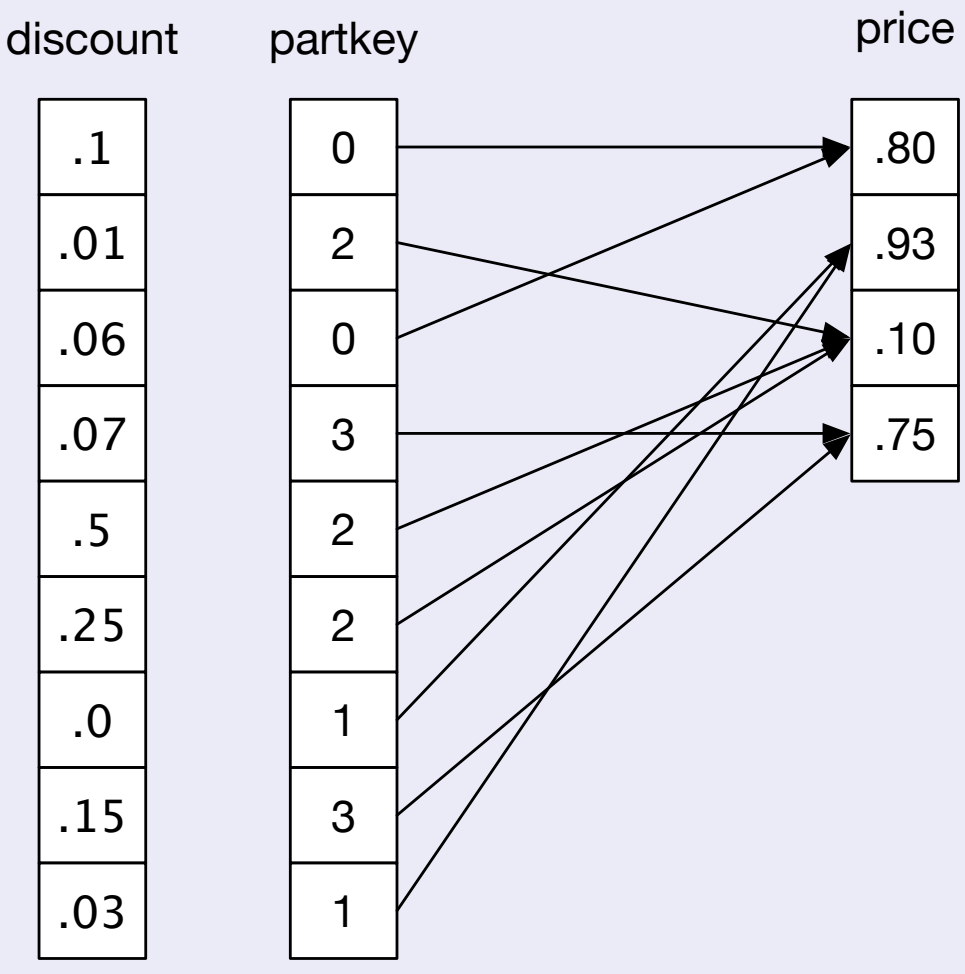
```
ALTER TABLE Orders
ADD FOREIGN KEY (BookID_index) REFERENCES Book(ID);
```

Idea

- Foreign Key (FK) constraints specify that
  - for every value that occurs in an attribute of a table
  - there is exactly one value in the Primary Key (PK) column of another table
- The DBMS needs to ensure that the constraint holds
  - On insert/update, the DBMS needs to look up the primary key value
  - Instead of storing the value, the DBMS could store a pointer to the referenced Primary Key or tuple
  - Is this a universally good idea?

Foreign-Key Indices

Example



Foreign-Key Indices

Usefulness

- The PK/FK constraint implies the number of join partners for every tuple: 1
- Resolving the FK reference column directly yields the join partner tuples
- Not of much use for anything else
  - However, many joins are PK/FK joins (because they stem from normalization)
- Foreign-Key Indices have very few downsides
  - Maintain insignificantly extra work under updates
  - They do not cost significant space (a pointer per tuple)
  - No extra query optimization effort: if they can be used, they should be
- SQL-Server does not implement them

B-Trees

B-Trees

Basic Idea

- Databases are I/O bound
  - $\rightarrow$ Minimize the number of page replacements
- There are many equality lookups
- There are also many updates
  - Hash-tables have nasty load-spikes on update
- Solution: Use a tree

Context

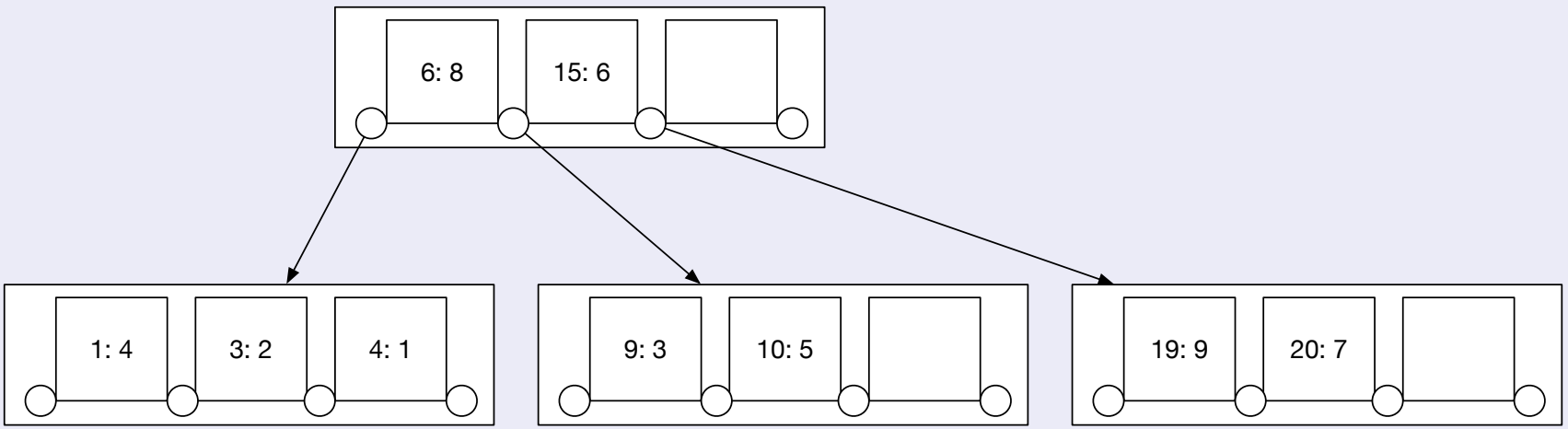
- In the DBMS context a Key/Value pair is the value of the attribute and a pointer to/id of the tuple

B-Trees

Definition

- A balanced tree with out-degree n (i.e., every node has $n-1$ keys) and the following property
- Each non-root node contains at least $\left\lfloor \frac{n-1}{2} \right\rfloor$ key/value pairs
- The root has at least one element

Example

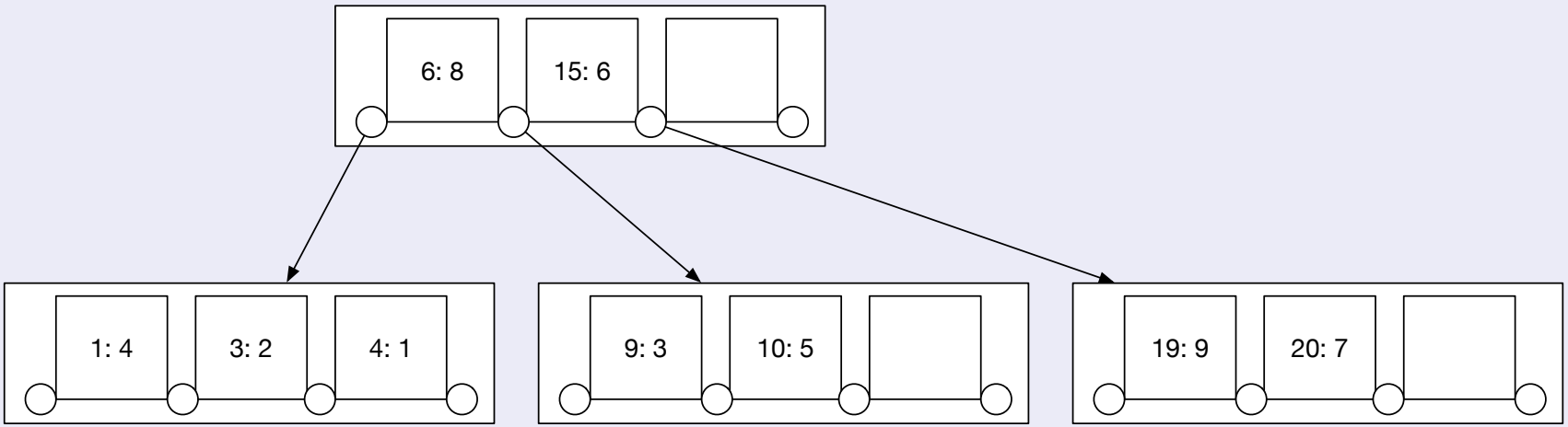


Maintaining balanced B-Trees

Under insert

- Find the right node to insert (walk the tree) and insert the value
- If the node overflows, split the node in two halves
- Insert a new split element (say, the one in the middle of the split-node) in the parent
- If the parent overflows, repeat the procedure on the parent node
  - If the parent is the root, introduce a new root

Example



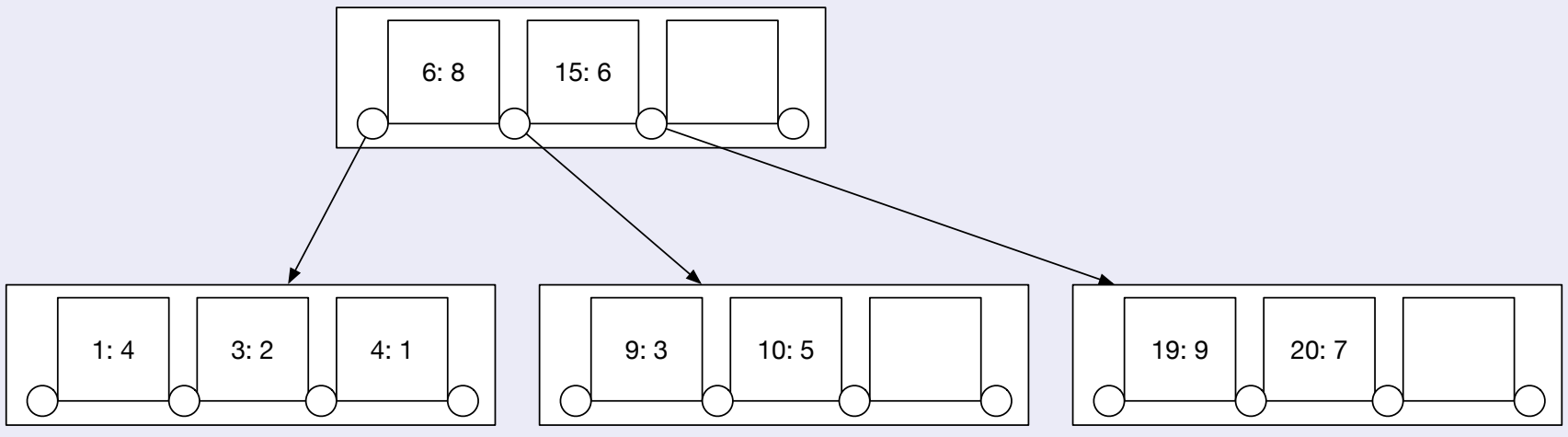
Maintaining balanced B-Trees

Under delete

- Find the value to delete
  - if it is in a leaf node, delete it
  - if it is in an internal node, replace it with the maximum leaf-node value from the left child
- If the affected leaf node underflows, rebalance the tree bottom up
  - Try to obtain an element from a neighbouring node (be done on success)
  - On failure, the neighbouring node is only half-full and can be merged with this one
  - merge and remove the parent spliting key
  - If parent underflows, rebalance from that one (bottom up)

Maintaining balanced B-Trees under delete

Example



Problems with B-trees

Access properties

- They can support range (`between 5 and 17`) scans but
  - it is complicated (need to go up and down the tree)
  - it causes many node traversals
  - Node sizes are usually co-designed with page sizes
  - Node traversals translate into page faults - we want to keep those to a minimum

Implementation complexity

- Two kinds of node layouts or space waste
  - Leaf pointers aren't used
  - Most of the data lives in leaf nodes

B^+-Trees

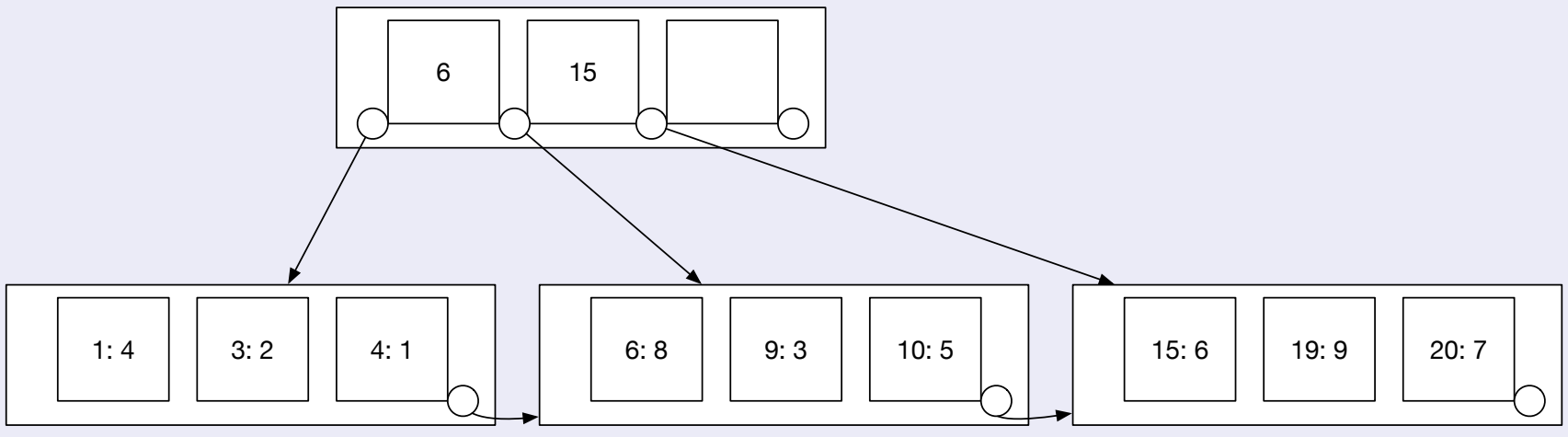
B^+-Trees

Idea

- Make range scans fast by
  - keeping data only in the leafs (no up and down)
  - linking one leaf to the next
  - inner-node split values are replicas of leaf-node values
- Only have a single kind of node layout

B^+-Trees

Example



B^+-Trees

Balancing

- Largely the same
- Deletes of inner-node split values imply replacement with new value from leaf node

Bitmap Indexing

Bitmap Indexing

Idea

- If there are few distinct values in a column
  - create a bitvector for each distinct value in that column

Bitvectors

- A sequence of 1-bit values indicating a boolean condition holding for the elements of a sequence of values
  - E.g., $BV_{==7}([4,7,11,7,7,11,4,7]) = [0,1,0,1,1,0,0,1]$
  - CPUs don't deal with individual bits – let's assume they deal with bytes (they do not)
  - $BV_{==7}([4,7,11,7,7,11,4,7]) =$
    $128 * 0 + 64 * 1 + 32 * 0 + 16 * 1 + 8 * 1 + 4 * 0 + 2 * 0 + 1 * 1 = 89$
  - Shoutout quiz: what does $BV_{==7} == 0$ mean?
  - Shoutout quiz: what does $BV_{==7} == 255$ mean?

Bitmap Indices

- A collection of bitvectors on a column

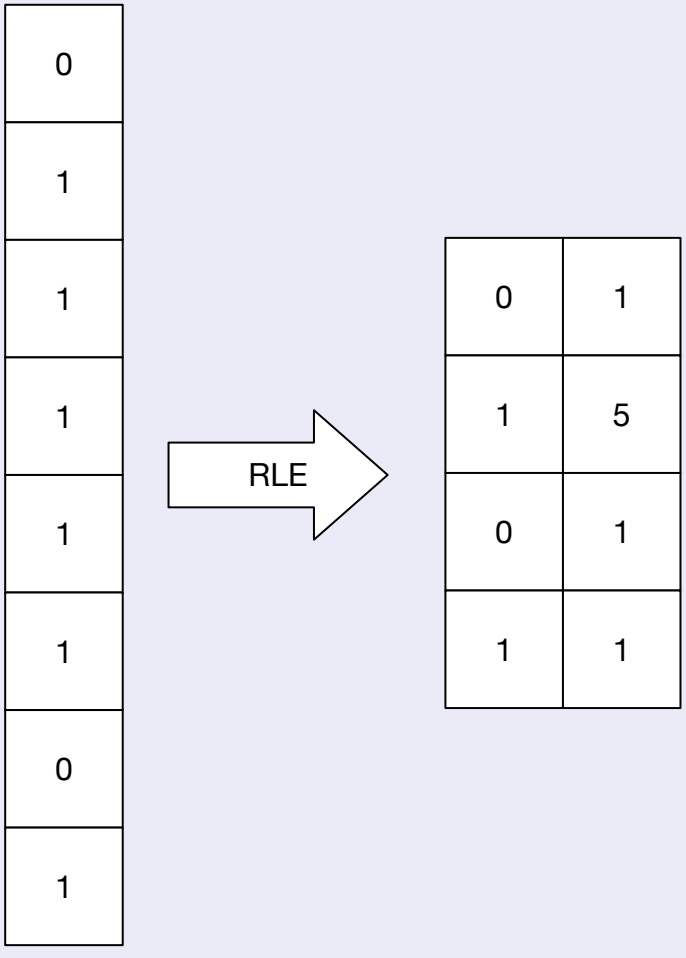
Bitmap Indexing

Usefulness

- Bitmaps reduce bandwidth need for scanning a column
  - in the order of the length of the type of the column
  - can be improved even further using simple compression (like run-length-encoding))
- Predicates can be combined using logical operators on bitvectors
- Arbitrary (boolean) conditions can be indexed by some systems
  - $BV_{>7,<12}([4,7,11,7,7,11,4,7]) =$
    $128 * 0 + 64 * 1 + 32 * 1 + 16 * 1 + 8 * 1 + 4 * 1 + 2 * 0 + 1 * 1 = 125$
- Special form: binned bitmaps
  - Have n bitvectors
  - Make sure the conditions span the entire value domain
    - In every position, exactly one value is set to one

Run-Length-Encoding

Visualisation



Description

- Sequentially traverse the vector
- Replace every run of consecutive equal values with
  - a tuple containing the value (*Run*) and the number of tuples (*length*)
- Works really well on high-locality data
- Requires sequential scan to find value at a specific position

Materialized Views

Materialized Views

A very quick note

- SQL allows "saving" queries as "views": `create view oids as (select Name, OrderID from Customer NATURAL JOIN Order)`
  - views can be used like relations
    - `select * from oids`
  - However, that makes no performance difference (the view name is nice syntax for a nested query)
- Some systems allow the creation of *Materialized Views*
  - These are actually stored (and maintained under updates)
- Some systems are very smart about using these internally even when users don't query them directly
  - see *Answering queries using views: A survey* by Alon Halevy

SQL Server

- Is weird about materialized views
  - You need to create a non-materialzed view first
  - and create an index on that

The End