Relational Database System Implementation

CS122 – Lecture 3

Winter Term, 2018-2019

Disk Records and Fields

- Tuples are ordered sets of attribute-value pairs
  - Every attribute has an associated type (a.k.a. "domain")
  - A value may also be **NULL** to represent unknown data
  - The data dictionary specifies the schema for every table
- Issues:
  - Can't expect a table to have all tuples be the same size
  - Also can't expect a table to have all non-**NULL** values
- Need a way to represent tuples within a disk page, where tuples can vary in size, and some attribute-values are unspecified

Disk Records and Fields (2)

- Fixed-size data types are easy to store into a tuple
  - e.g. **`INTEGER`**, **`CHAR(25)`**, **`DATE`** fields
  - Table's schema records each column's type
    - For columns with size/precision details, these are also stored
  - Just use schema to guide reading/writing the column
- Variable-size values also require a size to be stored
  - e.g. **`VARCHAR(`*`n`*`)`** fields
  - If $n < 256$: store 1-byte size, then string data
  - If $n < 65536$: store 2-byte size, then string data
  - (Can also terminate the field with a special character)

Disk Records and **NULL** Values

- In each tuple, include a bit for each attribute indicating whether its value is **NULL**
  - If bit is 1 then corresponding attribute has a **NULL** value
    - (Don't need to store data for **NULL** attributes in the record…)
  - Store bits in packed format: each byte holds 8 null-bits
  - Called a *null bitmap*
- Example record format:

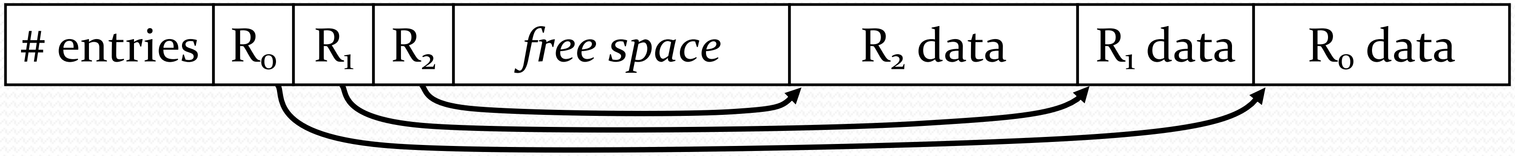
null bitmap	*user_id* (big-endian)	*username*	*name*	*website_url*
0x04	0xF0,0x95,0x01,0x00	0x06,'donnie'	NULL	0x22,'http://www.cs…'

  - (no data is actually stored for the *name* field)

Variable-Size Record Storage

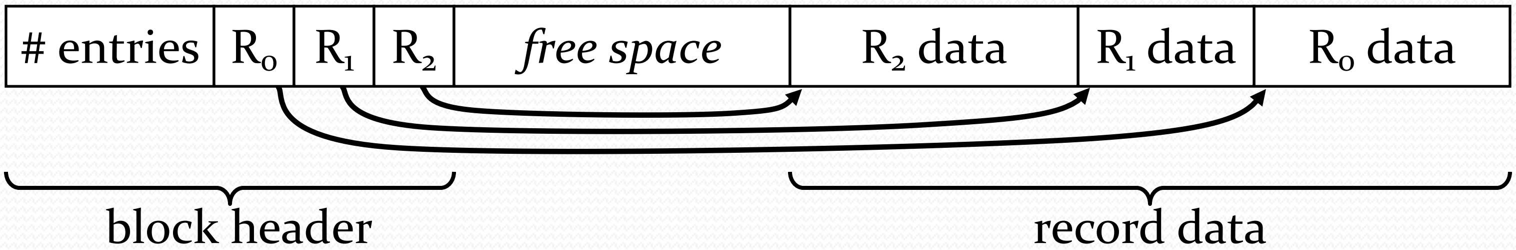
- Some row-values can vary in size
  - **`VARCHAR`**, **`BLOB`**, **`CLOB`**, **`TEXT`**, **`NUMERIC`**, etc. types
    - Some implementations of **`NUMERIC`** are fixed-size
  - Also, don't store any value for **`NULL`** fields
- Records will also vary in size

- Variable-size records can be stored into fixed-size blocks using a *slotted-page structure*



Slotted Page Structure (1)

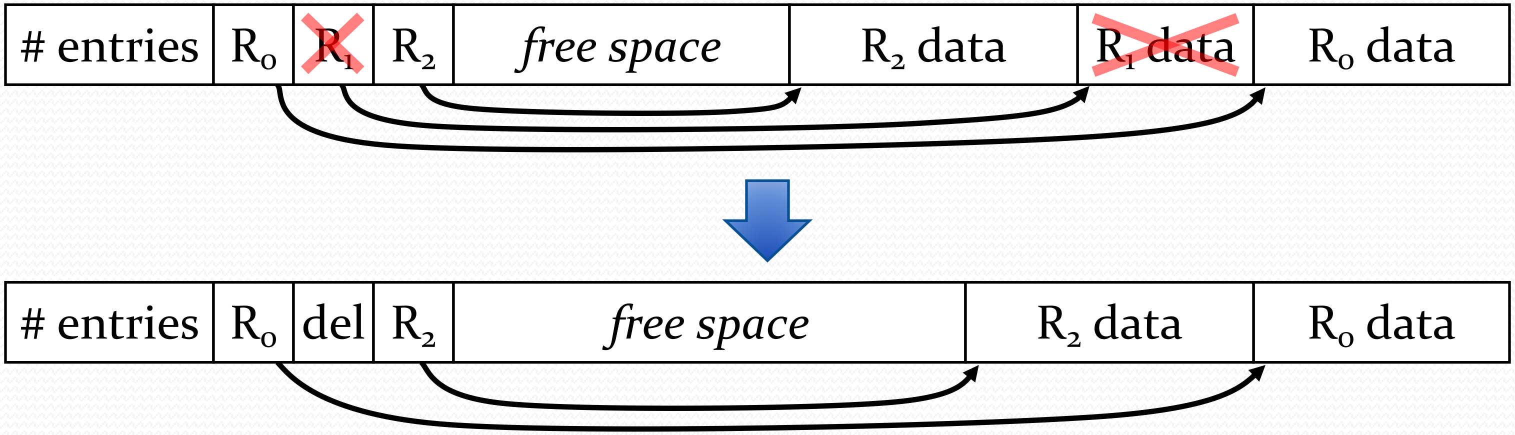
- The slotted-page structure:



- Records in a block are stored contiguously, starting from the *end* of the block
  - Records are stored in reverse order
- Start of block has a header specifying where each record in the block starts
  - First value specifies total number of records *N* in the block
  - Next *N* values specify the starting offset of each row's data

Slotted Page Structure (2)

- When a record is deleted:
  - Record's entry in the index is marked as "deleted"
    - (e.g. its index is set to an invalid value, such as 0)
  - The record's space is reclaimed within the block by moving other records toward end of block
- Example: Delete record 1 from this block:

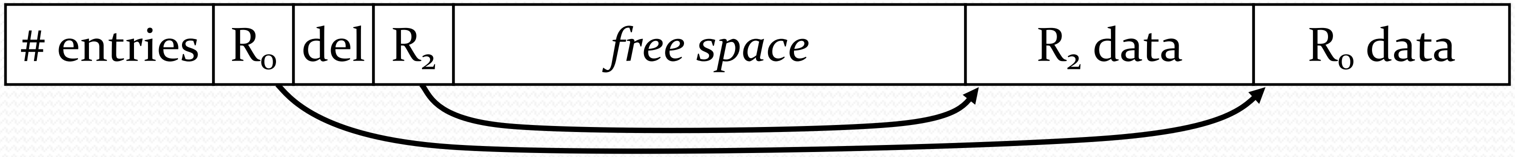


Indexes and Tables

- Table records may be referenced from other files
- Example:
  - Indexes allow specific rows to be found and retrieved, based on the values of some set of attributes
  - The index needs some way to reference a particular record
- Every record has a specific location in a data file:
  - The block the record is stored within
  - The offset of the record within the block
- Example: NanoDB record pointers:
  - Block number (unsigned short: 0 to 65535)
  - Offset within block (unsigned short: 0 to 65535)

Slotted Page Structure (3)

- With the slotted-page structure, records can be referenced by their index in the *block header*
  - Level of indirection allows record data to be moved within the block, without affecting data that references the record



- We can only shrink the slotted-page header when deleted records are at the *end* of the header area
  - e.g. cannot move entry R_2 to index 1 and shrink the header
  - When R_2 is deleted, then we can eliminate both entries
  - Or, if a new row is added to this block, it could occupy R_1

Record-Level File Organization

- Can also organize data files at the record-level
- *Heap file organization*
  - A record can appear anywhere within the data file
  - Very simple; requires little additional structure
  - Currently the most common file organization
- *Sequential file organization*
  - Records are stored in sequential order, based on a *search key*
- *Hashing file organization*
  - Records are stored in blocks based on a *hash key*
- *Multitable clustering file organization* – mentioned earlier

Sequential File Organization

- Records stored in sequential order based on *search key*
- If accessing the file based on the search key:
  - Sequential scan of the file produces records in sorted order
    - No additional work needed for producing sorted output
  - Can find individual records, or ranges of records, using binary search on the file
  - *(In many cases, also allows more efficient implementations of joins, grouping, and duplicate elimination)*
- If not accessing based on the search key:
  - Records are in no specific order
  - No different from accessing a heap file

Sequential File Organization (2)

- Search keys can contain multiple columns
- Given a table $T(A, B, C, D)$, with search-key (A, B, C):
  - Rows are ordered based on values of column A
  - Rows with the same value of column A are ordered on B
  - etc.
  - If table is sorted on (A, B, C), it is also sorted on (A) and (A, B)
- If a query needs rows from T in order of (A) or (A, B), again no sorting is required!

Sequential File Organization (3)

- How do we maintain sequential order of records?
  - How to insert new records into sequential file?
  - What about deleting records?
  - Clearly, rearranging the entire file is unacceptable

- A simple (naïve) implementation strategy:
  - Add a pointer to each record, specifying next record in the file

Sequential Files

- Example:
  - Accounts, ordered by branch name
  - Initially, each record pointer references the next record
- When new record is added
  - If block containing previous record has space for a new record, add it there
  - Otherwise, append record to end of file
  - Update pointer chain to reflect new record order

A-217	Brighton	750	•
A-101	Downtown	500	•
A-110	Downtown	600	•
A-215	Mianus	700	•
A-102	Perryridge	400	•
A-201	Perryridge	900	•
A-218	Perryridge	700	

A-217	Brighton	750	•
A-101	Downtown	500	•
A-110	Downtown	600	•
A-215	Mianus	700	•
A-102	Perryridge	400	•
A-201	Perryridge	900	•
A-218	Perryridge	700	
A-888	North Town	700	•

Sequential File Organization (4)

- Ideally, key order and physical layout will match closely
  - Could maintain extra space in blocks to help keep nearby tuples in the same (or nearby?) blocks
  - After many inserts and deletes, file will eventually become disorganized
- Without maintenance, sequential scans or binary searches would eventually become *very* expensive
  - Disk seek time would kill performance
  - *(SSD would avoid this problem!)*
- Must periodically reorganize the file to ensure physical order of records matches key order
  - (Could do this when system load is typically low)

Hashing File Organization

- Records are stored in a location based on a *hash key*
- If accessing the file based on the hash key:
  - Very fast for finding records with a specific value
  - Doesn't support general inequality comparisons, ranges, etc.!
    - Really only good for equality comparisons
- If not accessing based on the hash key:
  - Again, records are in no specific order
  - No different from accessing a heap file
- As before, hash key can contain multiple columns
  - Unfortunately, far less useful than search keys with multiple columns

Hashing File Organization (2)

- In-memory hash tables:
  - Can use a fixed number of bins with overflow chaining, or open addressing, to handle placement of entries
  - As the table becomes full, it must periodically be reorganized
  - Increase number of locations, and spread out the entries
- How do we manage a hash table of records <u>in a file</u>?
  - Again, rearranging the entire file would be unacceptable

Static Hashing

- Generally, open addressing isn't well suited to data files
- Create some number of buckets to store records
  - Use overflow chaining when a bucket is full
- A simple solution: *static hashing*
  - Create a <u>fixed</u> number of buckets B
    - Different ways to represent buckets in the data file
    - e.g. each bucket is one disk block, or N sequential disk blocks
  - Hash key k is mapped to a bucket b with a hash function $h(k)$
  - Store each record into the bucket specified by the hash function

Static Hashing (2)

- Devote part of file to mapping from bucket # to block #
  - e.g. block 0 holds mapping
- If bucket holds any records, entry specifies block number where records are stored
  - Otherwise, use some value to indicate an empty bucket
- As records are added to file, assign blocks to buckets as needed

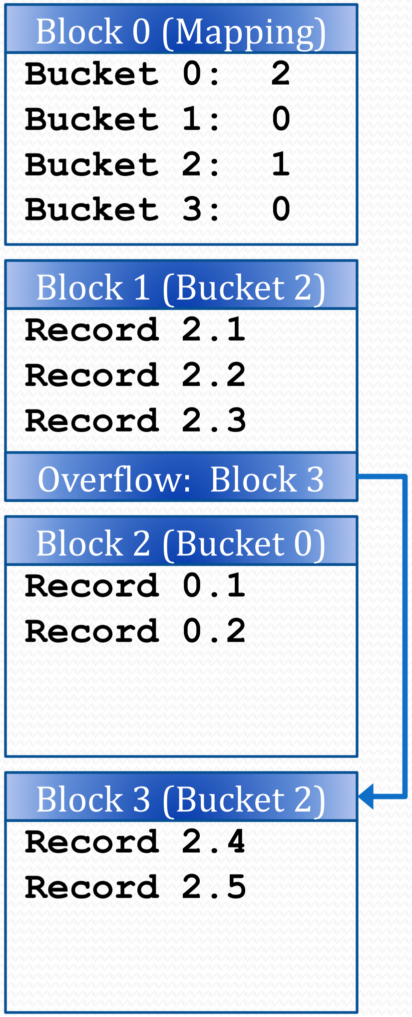
Block 0 (Mapping)	
Bucket 0:	2
Bucket 1:	0
Bucket 2:	1
Bucket 3:	0

Block 1 (Bucket 2)
Record 2.1
Record 2.2
Record 2.3

Block 2 (Bucket 0)
Record 0.1
Record 0.2

Static Hashing (3)

- If a bucket becomes full, must overflow records into another location!
- Several options for managing overflow records
  - e.g. create linked chains of blocks, as before
- If a record is deleted from a chain of blocks, can move records from overflow blocks into earlier blocks



Static Hashing (4)

- Static hashing has some big limitations:
- Data files frequently grow in size over their lifetime
  - Must predict how many buckets are necessary at start
  - If buckets end up being too full, lookups will involve lots of scanning through overflow blocks
- May end up with data that doesn't hash well!
  - e.g. data doesn't have a good distribution for the number of buckets, or if the hash function isn't very good
  - Again, end up with some buckets that hold many records
- Would prefer a *dynamic hashing* mechanism
  - Allow the number of buckets to change over time, without requiring the entire data file to be reorganized

File Organization: Summary

- Simplest file organization is heap file organization
  - No particular order for records in the file
  - Requires no additional record-level organization
- Other file organizations can dramatically improve access performance, but only in specific situations!
  - Can use alternate organization to make queries fast…
  - If query doesn't match file organization's characteristics, it's equivalent to accessing a heap file
- If physical organization doesn't correspond to logical organization, access can be *very* slow
  - e.g. increased disk seeks for out-of-order sequential file

File Organization: Summary (2)

- If a sequential or hash file changes frequently, periodic reorganization may be required
  - Will likely require moving large numbers of records
- Most common solution:
  - Store the records themselves in a heap file
  - Build one or more *indexes* into the heap file
    - Indexes are generally either ordered (typical) or hashed
    - Indexes reference records in heap file using record pointers
  - Index entries are much smaller than table records:
    - Can fit many more into each disk block
    - Much faster to move and reorganize them

File Organization: Summary (3)

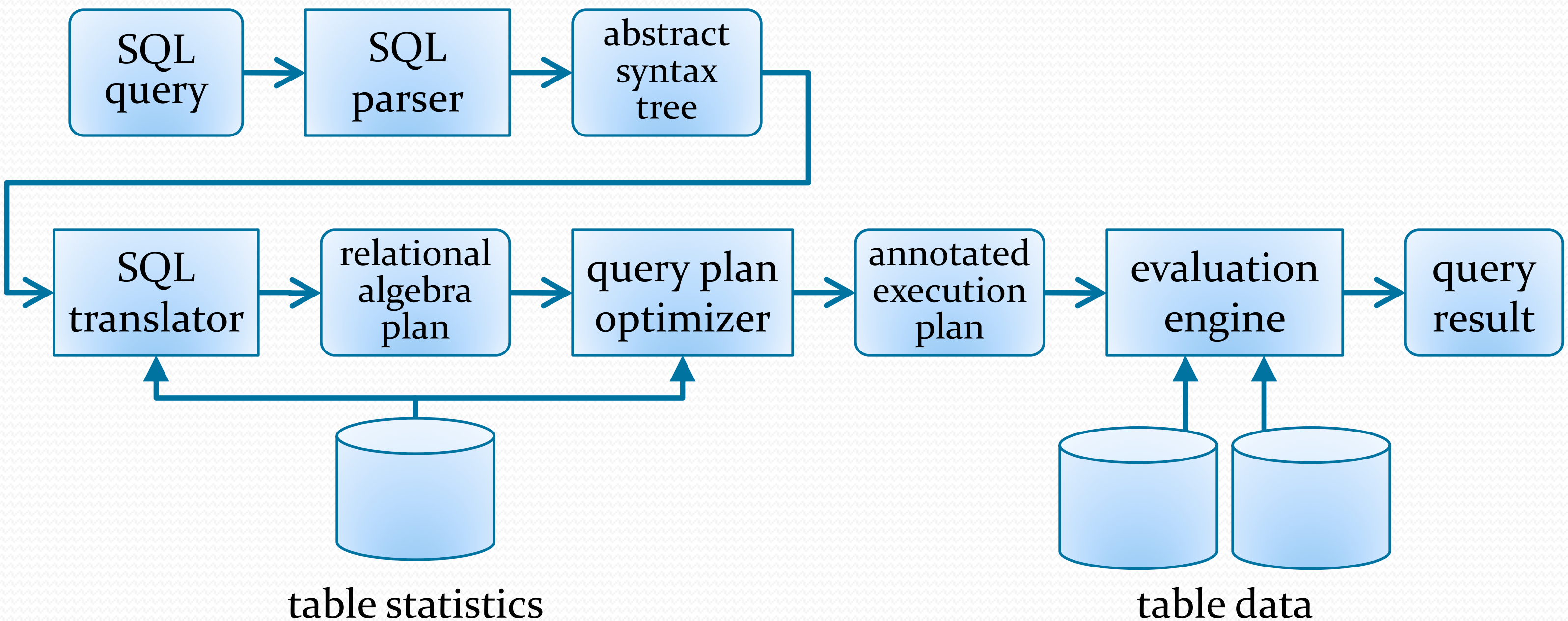
- When we are evaluating a query:
  - If we can, utilize indexes to do faster lookups in heap file
  - (Or, just evaluate query against the index!)
  - If not, just do a sequential scan through the heap file
- Will talk much more about indexes in a few weeks!
- For now, just focus on queries against heap files

SQL Query Evaluation

- Relational databases frequently use SQL query language to specify queries
- Databases don't execute SQL directly!
  - Very complicated language
  - Difficult to transform/optimize before executing
- SQL is transformed into a plan based on the relational algebra, and then executed by the query evaluator
- First step is to translate SQL into an abstract syntax tree
- In NanoDB, top-level object is a Command
  - Subclasses for various commands, e.g. CreateTableCommand
- If command is a DDL operation, it is executed directly

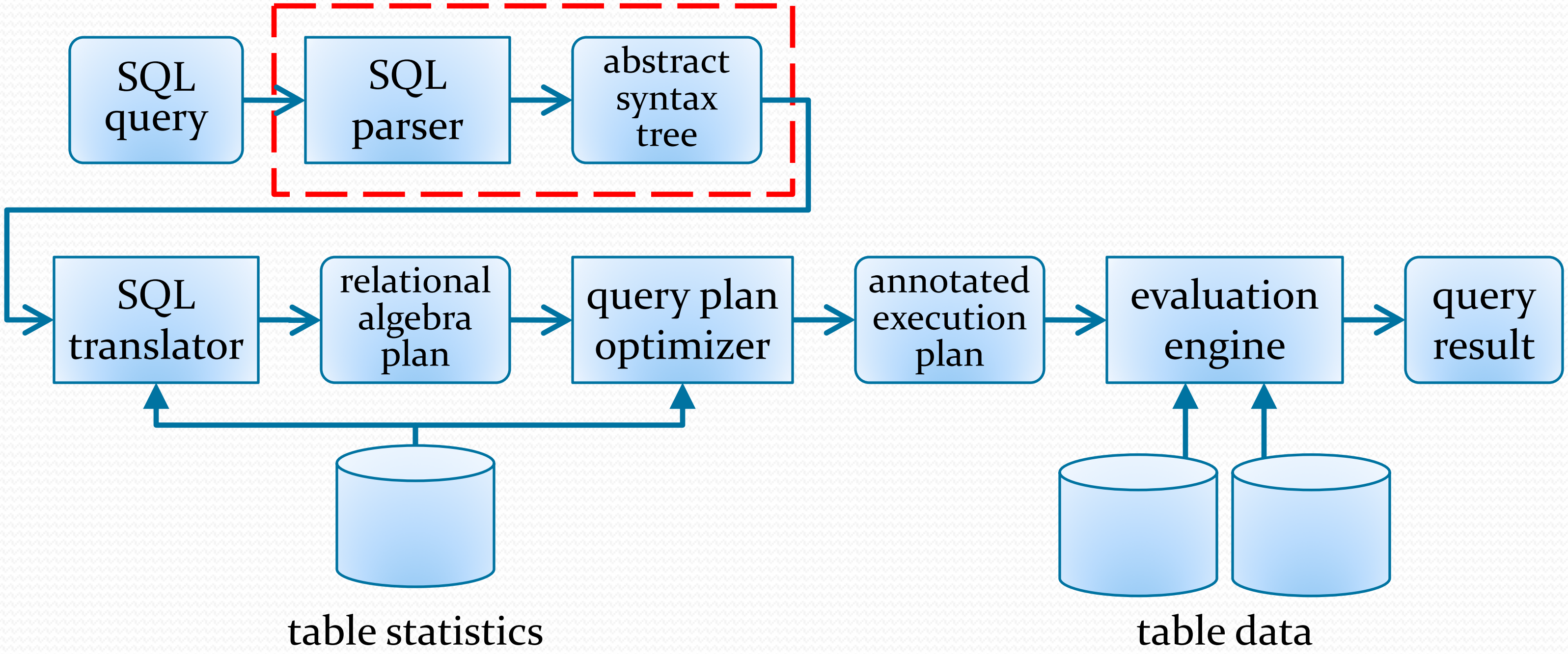
Query Evaluation Pipeline

- DML operations are processed through these stages:
  - e.g. SELECT, INSERT, UPDATE, DELETE



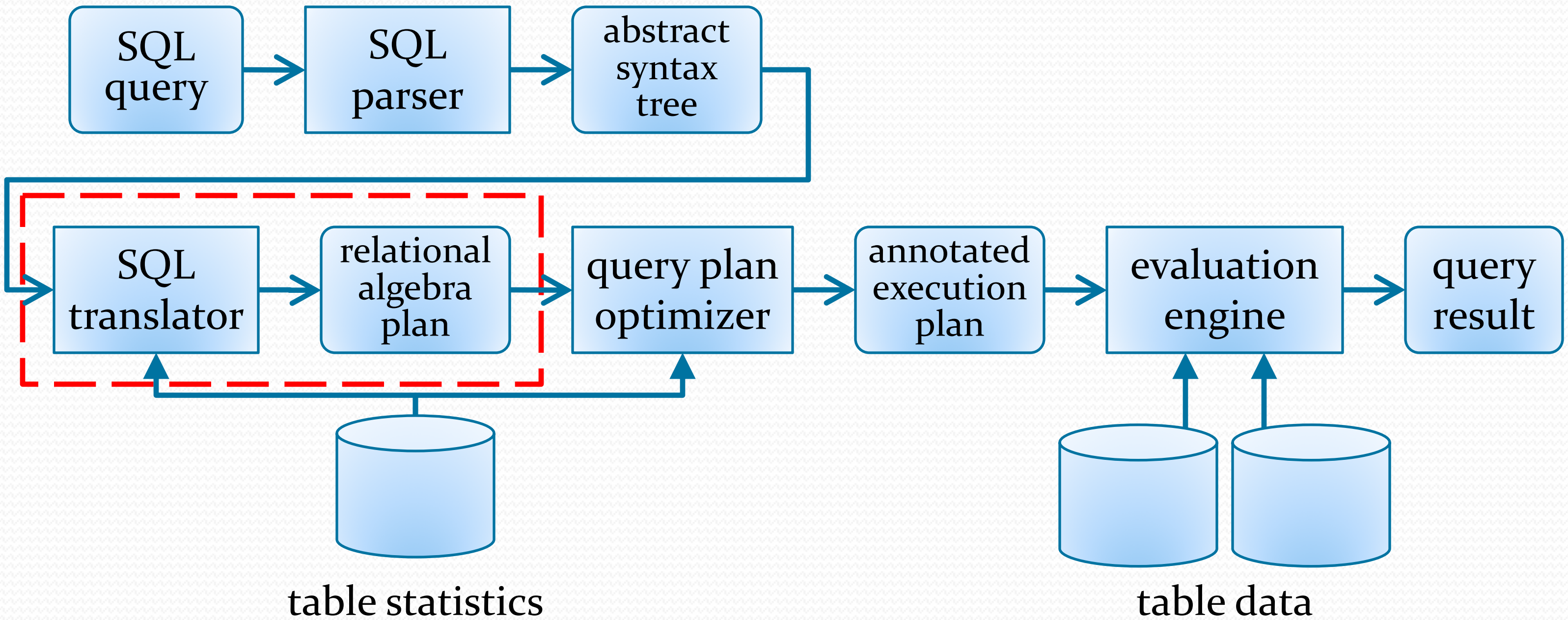
Query Evaluation Pipeline (2)

- SQL queries are parsed into an abstract syntax tree
  - AST represents the query as a hierarchy of related SELECT-FROM-WHERE operations
  - Sometimes called "SFW blocks"



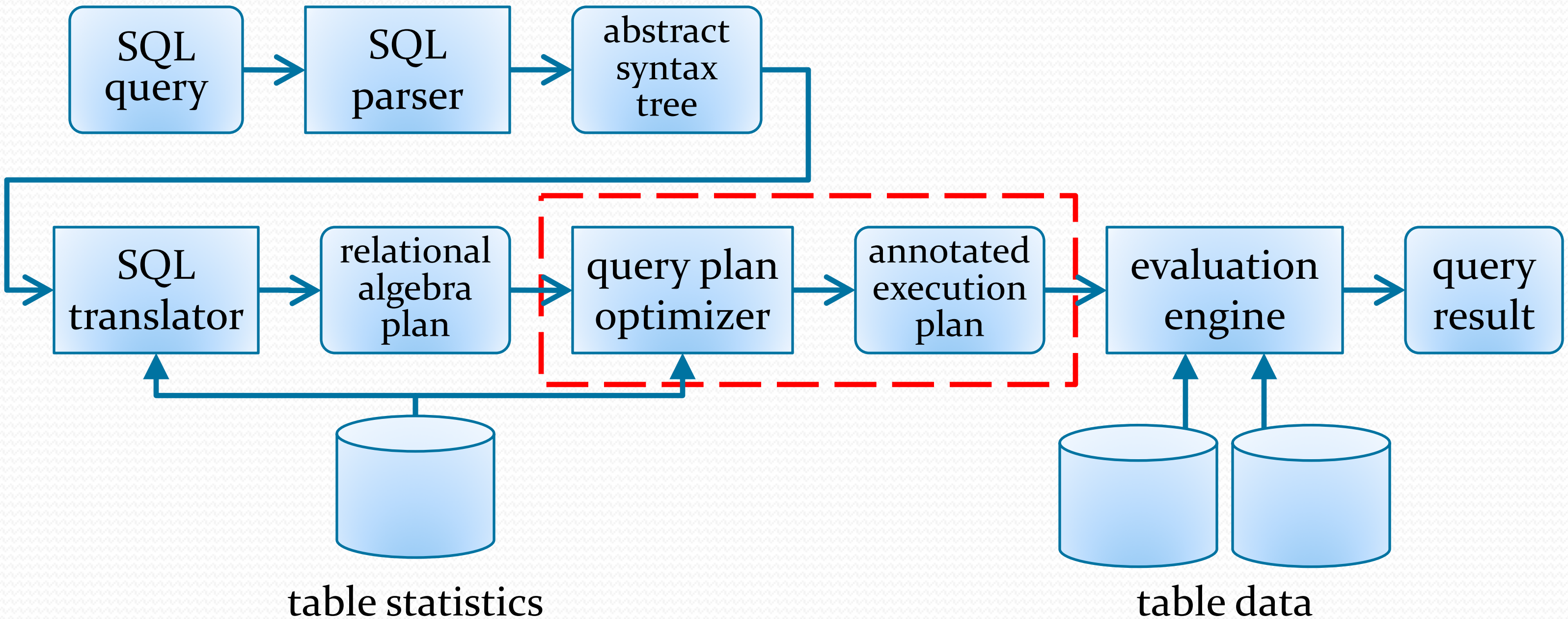
Query Evaluation Pipeline (3)

- Query AST is then translated into an initial query plan
  - Plan is based on relational algebra operations
  - Can apply some high-level optimizations to the AST
  - Also, join ordering can be determined in this phase



Query Evaluation Pipeline (4)

- Initial query plan is then optimized
  - Optimizer applies additional optimizations to plan
  - Determines final execution details for each plan node
    - e.g. best algorithm to use, which indexes to use, etc.



Query Evaluation Pipeline (5)

- Finally, execution plan is evaluated against the tables!
  - At this point, operation is generally very straightforward