Relational Database System Implementation

CS122 – Lecture 3

Winter Term, 2017-2018

Record-Level File Organization

- Last time, finished discussing block-level 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!

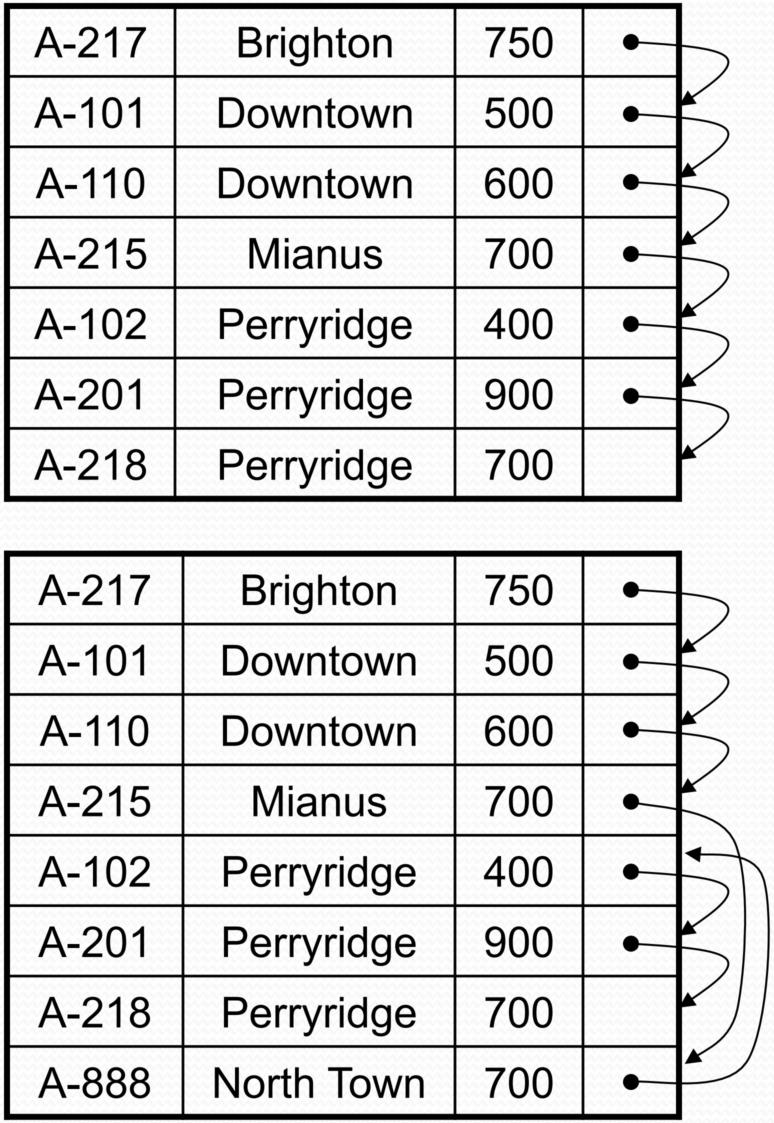
Sequential File Organization (3)

- 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 in a file?
  - 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)

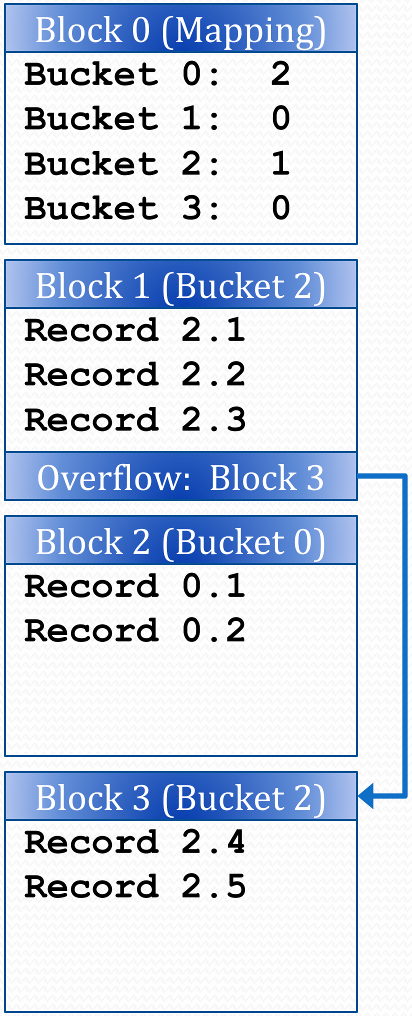
```
Record 2.1
Record 2.2
Record 2.3
```

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



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`
Overflow: Block 3

Block 2 (Bucket 0)
`Record 0.1`
`Record 0.2`

Block 3 (Bucket 2)
`Record 2.4`
`Record 2.5`

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 heap 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

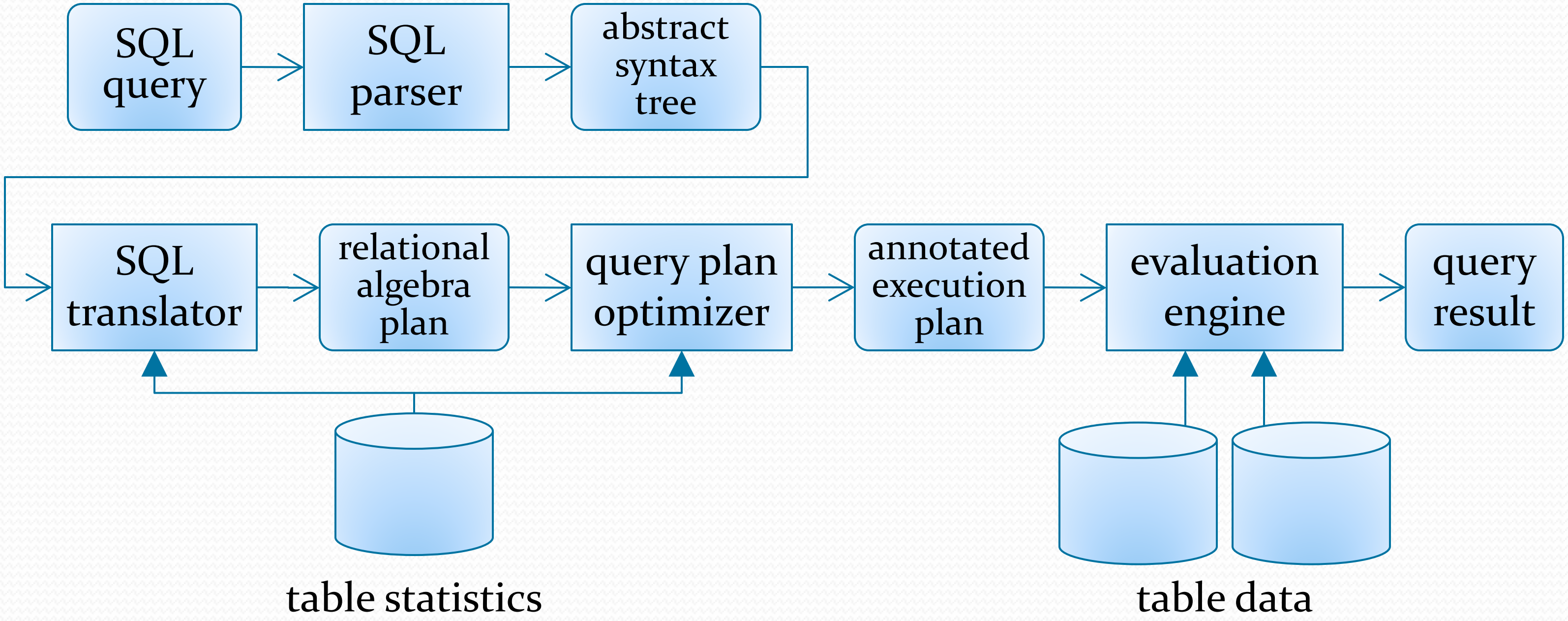
- Will talk much more about indexes in a few weeks!
- For now, just focus on queries against heap files

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

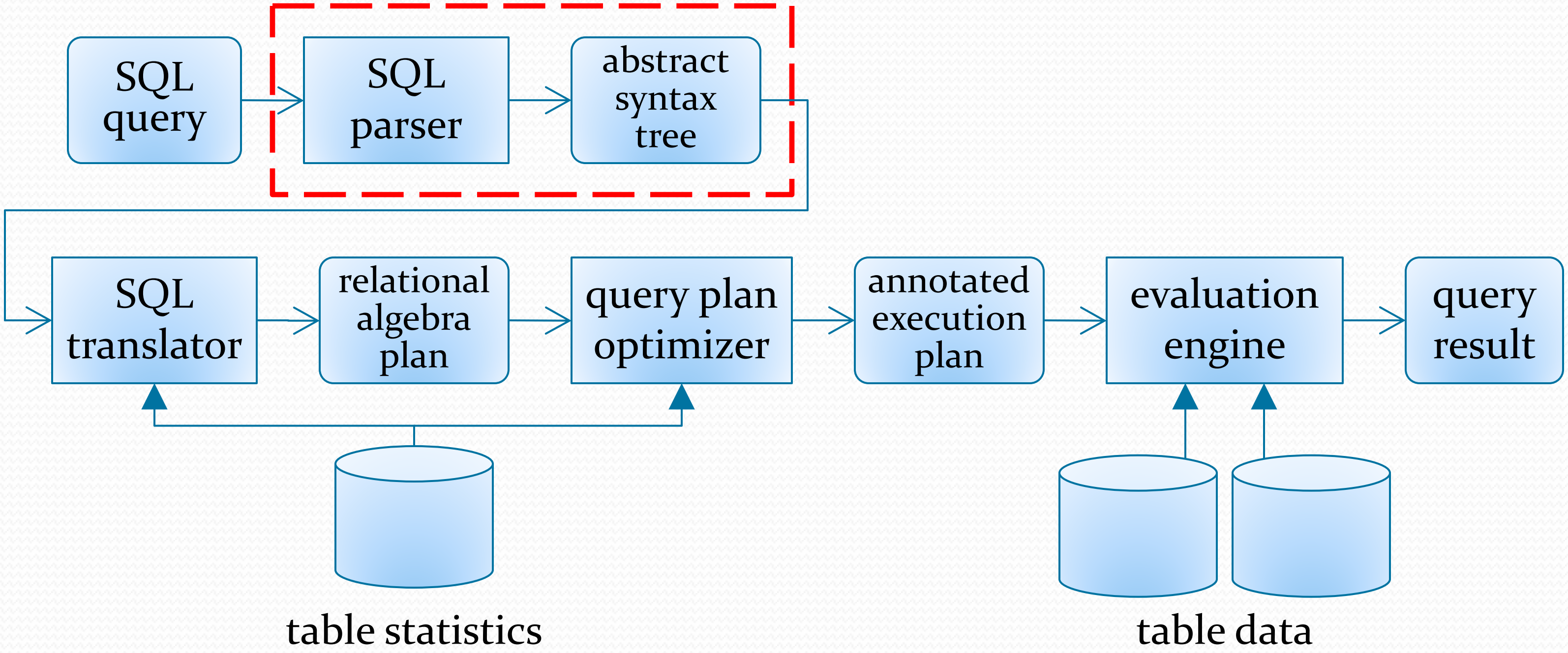
Query Evaluation Pipeline

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



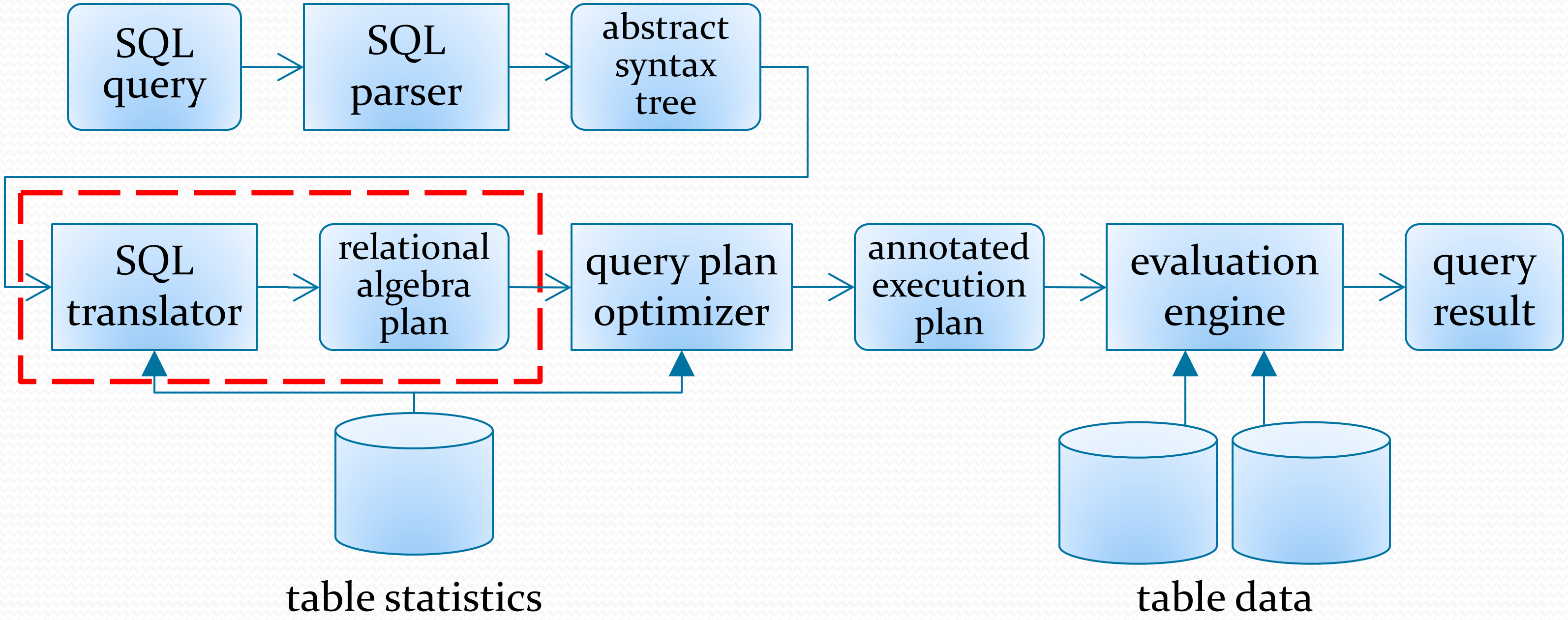
Query Evaluation Pipeline (2)

- 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"



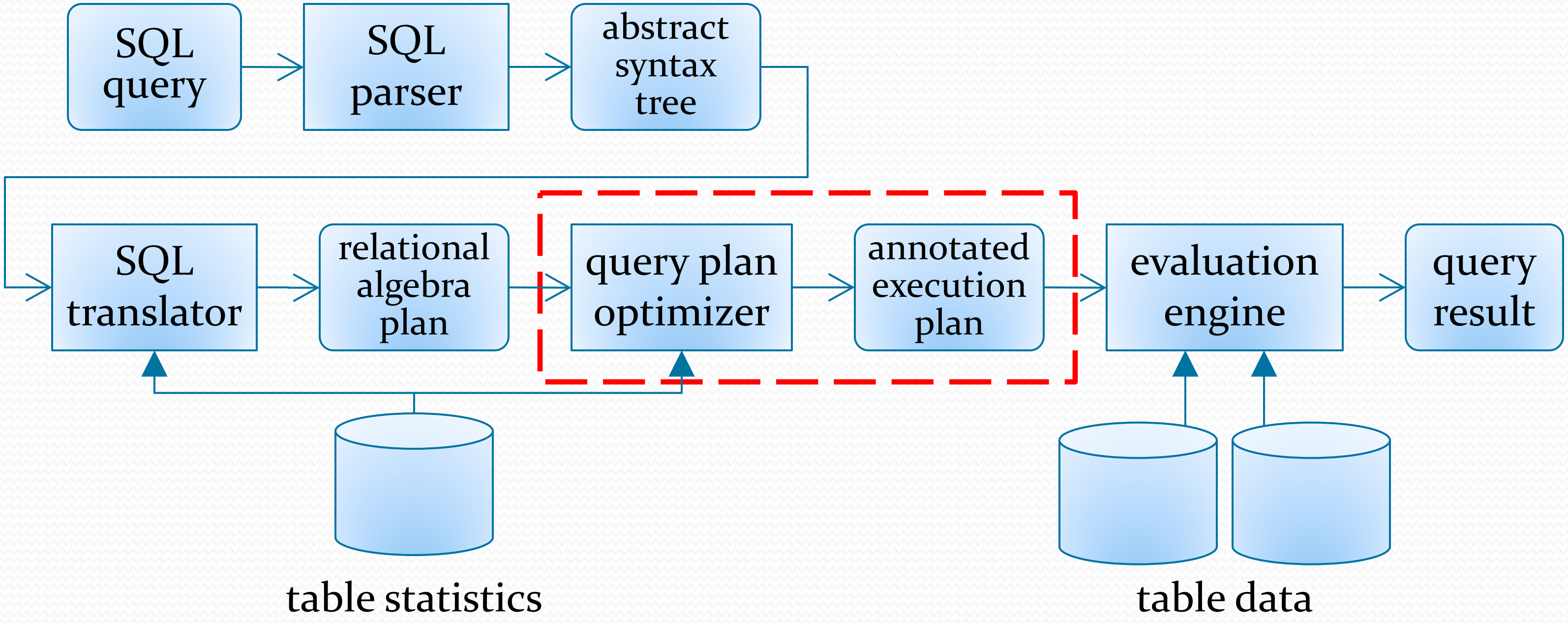
Query Evaluation Pipeline (3)

- 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



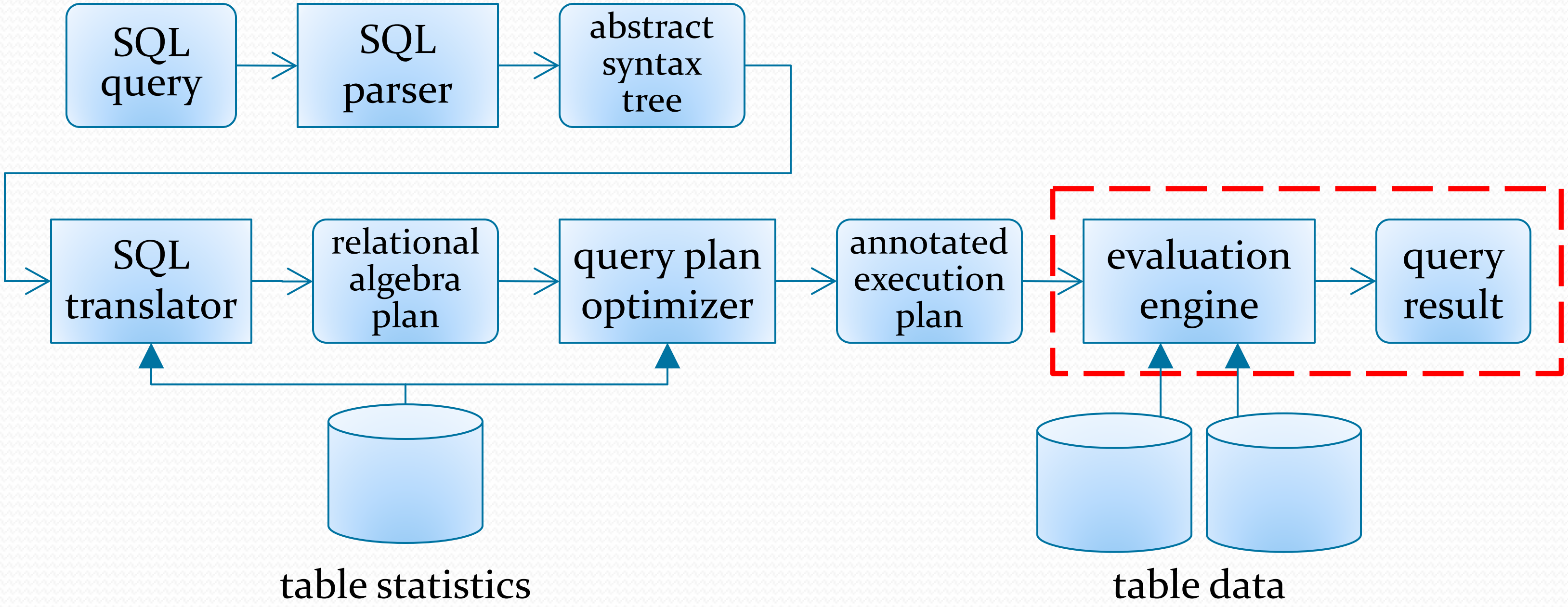
Query Evaluation Pipeline (4)

- 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



SQL Data Manipulation

- Can handle SELECT, INSERT, UPDATE, DELETE all with same evaluation pipeline
- A good idea anyway, since INSERT, UPDATE, DELETE can all have subqueries in them!

```
INSERT INTO t1 (a, b, c)
    SELECT a, b + 2, c – 5 FROM t2 WHERE d > 5;
UPDATE t1 SET a = a + 5
    WHERE c IN (SELECT c FROM t2);
UPDATE t1 SET a = (SELECT a FROM t2 WHERE t1.b = t2.b);
DELETE FROM t1
    WHERE a = (SELECT MAX(a) FROM t2 WHERE t1.b = t2.b);
```

SQL Data Manipulation (2)

- All four statements generate a set of tuples…
  - Only difference is what we do with them.
  - SELECT selects tuples for display/transmission to client
  - INSERT selects tuples for insertion into a table
  - UPDATE selects tuples for modification
  - DELETE selects tuples for removal
- NanoDB query evaluator takes an execution plan, and a tuple-processor that handles the results
  - For each tuple produced by the execution plan, the tuple-processor does something with the tuple
  - e.g. the TupleUpdater modifies the tuple based on the UPDATE statement issued to the database

SQL Data Manipulation (3)

```
EvalStats QueryEvaluator.executePlan(
    PlanNode plan, TupleProcessor processor)
```

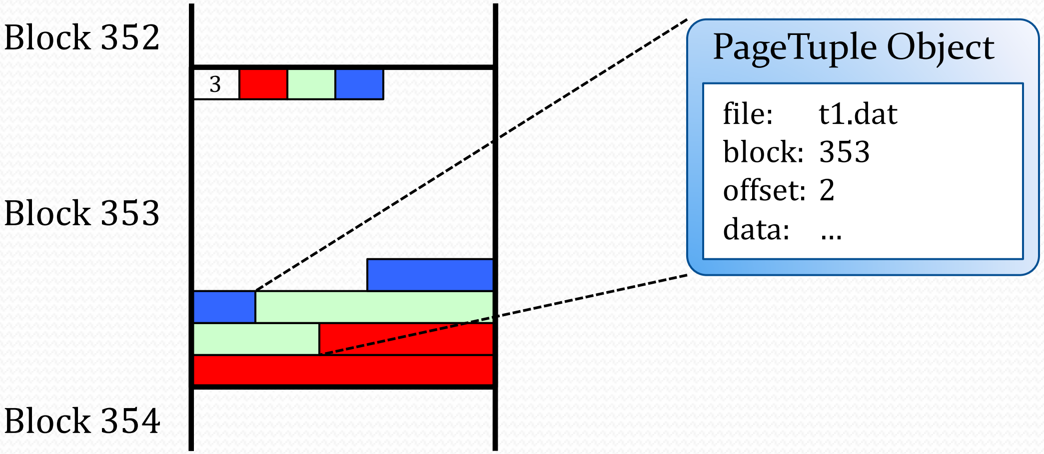
- Evaluator also returns statistics about the evaluation
  - Databases generally tell you how many rows were selected/inserted/updated/deleted, and how long the query took
- **Not all tuples are created equal!**
  - Some tuples can simply be displayed or sent to client
  - Some tuples must support modification or deletion
  - Databases also have a notion of “l-values” and “r-values”

L-Values and R-Values

- Only certain expressions can be used on the left-hand side of an assignment operation
- Example: **`a = 5 + b * 3;`**
  - **`a`**, **`b`**, **`5`** and **`3`** are all values
  - Only some of these can be the target of an assignment
- L-values are values with an associated location/address
  - Knowing the location allows us to modify the value
  - “L” indicates it can appear on left-hand side of an assignment
- R-values don’t have a location
  - i.e. the value cannot be a target of an assignment operation
  - “R” indicates it must be on right-hand side of the assignment

Kinds of Tuples

- Different flavors of tuples in a database engine
- Some tuples are backed by a page in a database table
  - Reading values from tuple come straight from data page
  - Writing to the tuple modifies the data page in memory
  - (page must then be flushed back to disk)



Kinds of Tuples (2)

- Other tuples contain computed values, and are stored in memory only
  - This query generates computed results:

    SELECT username, SUM(score) AS total_score
    FROM game_scores GROUP BY username;
  - NanoDB represents these as TupleLiteral objects

- Many database implementations represent all tuples in the same format, in memory buffers
  - Allows them to be written to disk very easily, if needed

Kinds of Tuples (3)

- SELECT and INSERT…SELECT statements don’t require lvalue tuples
  - Results are either displayed, or added to a data file
- UPDATE and DELETE <u>require</u> lvalue tuples
  - Selected tuples are modified or removed!
    - Actually modifies a data file
  - Plans generated for UPDATE and DELETE must take this into account
  - Constrains the optimizations that may be employed