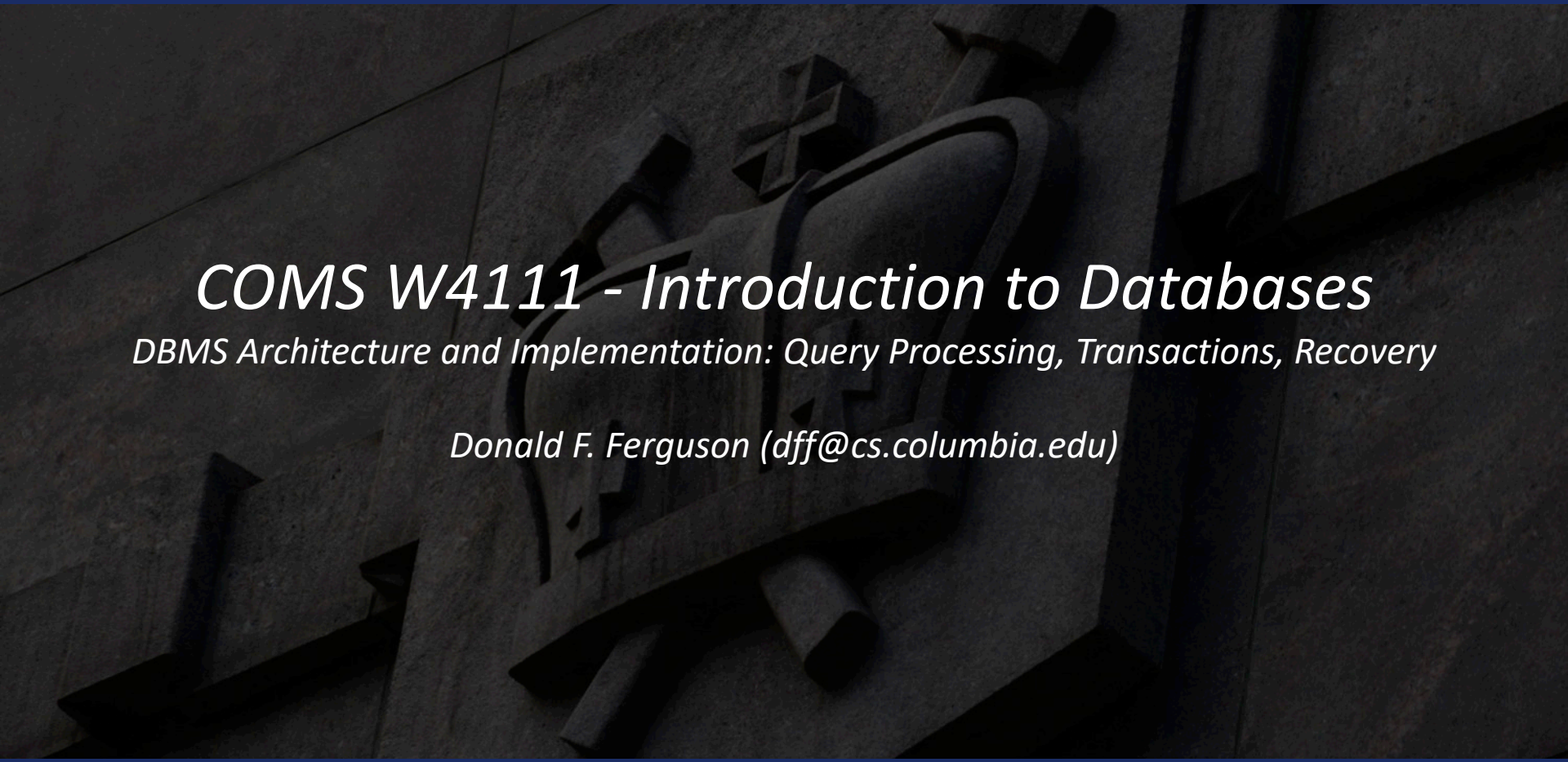


COMS W4111 - Introduction to Databases

DBMS Architecture and Implementation: Query Processing, Transactions, Recovery

Donald F. Ferguson (dff@cs.columbia.edu)

Query Processing

Overview

Query Compilation

Database Systems: The Complete Book (2nd Edition) 2nd Edition
by Hector Garcia-Molina (Author), Jeffrey D. Ullman (Author), Jennifer Widom (Author)

Preview of Query Compilation

To set the context for query execution, we offer a very brief outline of the content of the next chapter. Query compilation is divided into the three major steps shown in Fig. 15.2.

a) *Parsing.* A *parse tree* for the query is constructed.

b) *Query Rewrite.* The parse tree is converted to an initial query plan, which is usually an algebraic representation of the query. This initial plan is then transformed into an equivalent plan that is expected to require less time to execute.

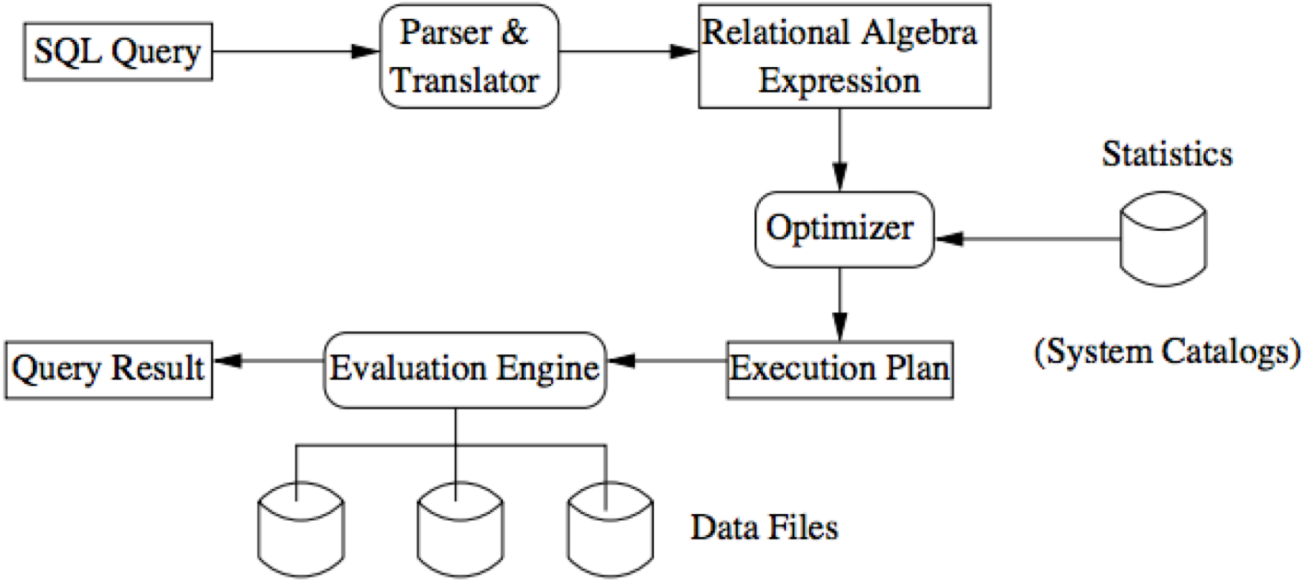
c) *Physical Plan Generation.* The abstract query plan from (b), often called a *logical query plan*, is turned into a *physical query plan* by selecting algorithms to implement each of the operators of the logical plan, and by selecting an order of execution for these operators. The physical plan, like the result of parsing and the logical plan, is represented by an expression tree. The physical plan also includes details such as how the queried relations are accessed, and when and if a relation should be sorted.

Parsing and Execution

- Parser/Translator
  - Verifies syntax correctness and generates a *parse tree*.
  - Converts to *logical plan tree* that defines how to execute the query.
    - Tree nodes are *operator(tables, parameters)*
    - Edges are the flow of data "up the tree" from node to node.
- Optimizer
  - Modifies the logical plan to define an improved execution.
  - Query rewrite/transformation.
  - Determines *how* to choose among multiple implementations of operators.
- Engine
  - Executes the plan
  - May modify the plan to *optimize* execution, e.g. using indexes.

Query Processing Overview

Basic Steps in Processing an SQL Query



Parsing

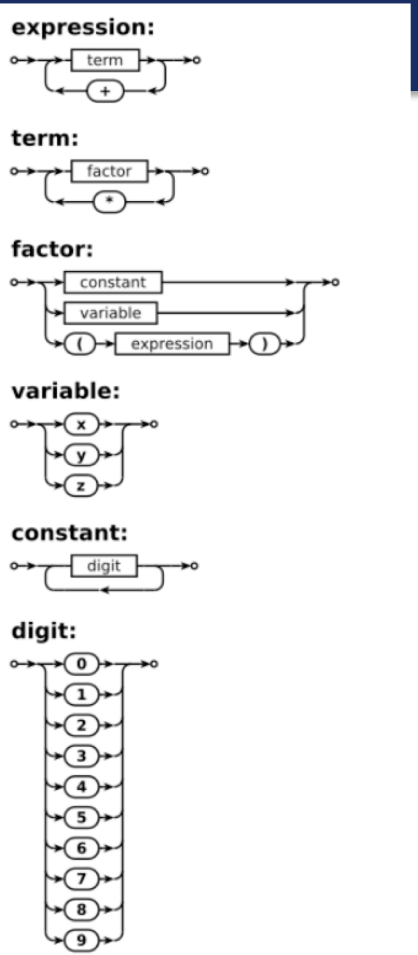
Grammar/Syntax Diagrams

We use arithmetic expressions as an example. First we provide a simplified BNF grammar:

```
<expression> ::= <term> | <expression> "+" <term>
<term>       ::= <factor> | <term> "*" <factor>
<factor>     ::= <constant> | <variable> | "(" <expression> ")"
<variable>   ::= "x" | "y" | "z"
<constant>   ::= <digit> | <digit> <constant>
<digit>      ::= "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9"
```

This grammar can also be expressed in EBNF:

```
expression = term | expression, "+" , term;
term       = factor | term, "*" , factor;
factor     = constant | variable | "(" , expression , ")";
variable   = "x" | "y" | "z";
constant   = digit , {digit};
digit      = "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9";
```

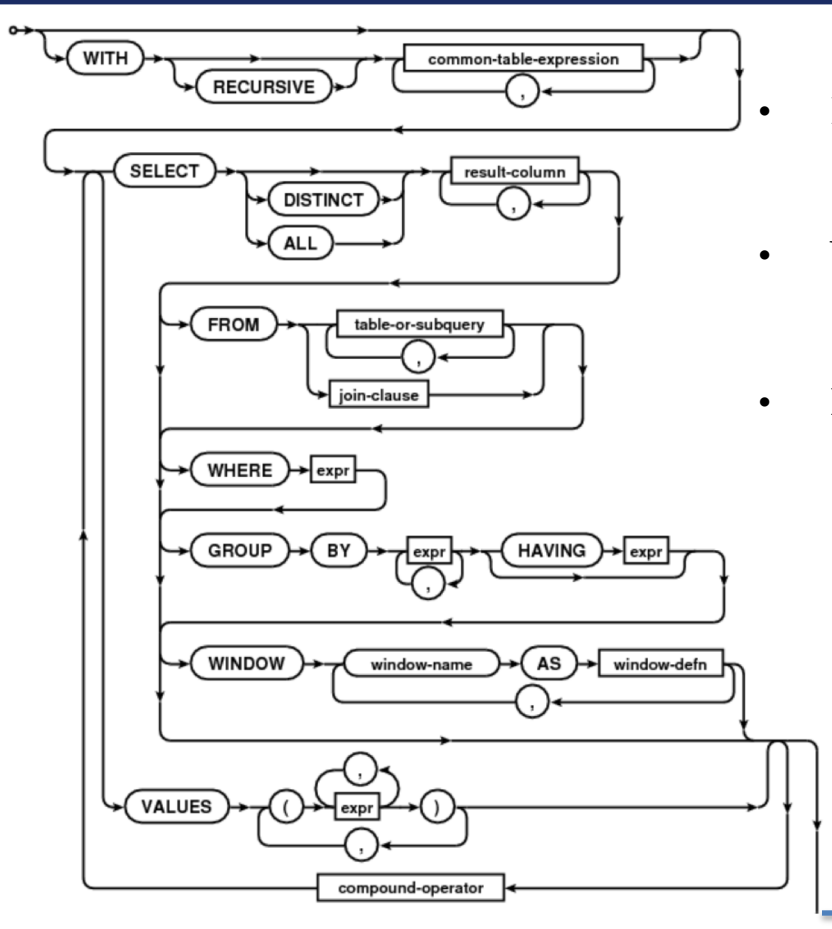


Parser

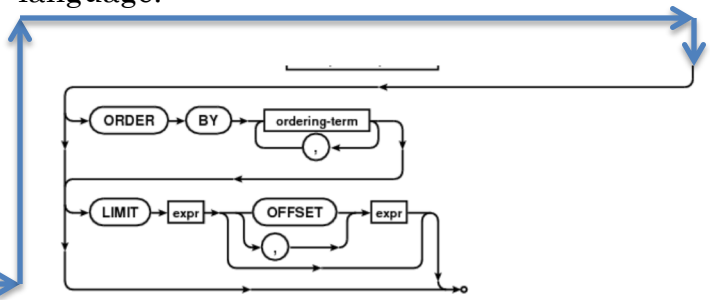
- Grammar rules define valid SQL statements.
- Query parser
  - Validates syntax.
  - Produces a logical parse tree/plan.
- The optimizer and execution engine execute the plan.
- Similar to any programming language

```
SELECT
    [ALL | DISTINCT | DISTINCTROW ]
      [HIGH_PRIORITY]
      [STRAIGHT_JOIN]
      [SQL_SMALL_RESULT] [SQL_BIG_RESULT] [SQL_BUFFER_RESULT]
      SQL_NO_CACHE [SQL_CALC_FOUND_ROWS]
    select_expr [, select_expr ...]
    [FROM table_references
      [PARTITION partition_list]
    [WHERE where_condition]
    [GROUP BY {col_name | expr | position}, ... [WITH ROLLUP]]
    [HAVING where_condition]
    [WINDOW window_name AS (window_spec)
        [, window_name AS (window_spec)] ...]
    [ORDER BY {col_name | expr | position}
      [ASC | DESC], ... [WITH ROLLUP]]
    [LIMIT {[offset,] row_count | row_count OFFSET offset}]
    [INTO OUTFILE 'file_name'
        [CHARACTER SET charset_name]
        export_options
      | INTO DUMPFILE 'file_name'
      | INTO var_name [, var_name]]
    [FOR {UPDATE | SHARE} [OF tbl_name [, tbl_name] ...] [NOWAIT | SKIP LOCKED]
      | LOCK IN SHARE MODE]]
```

Parser

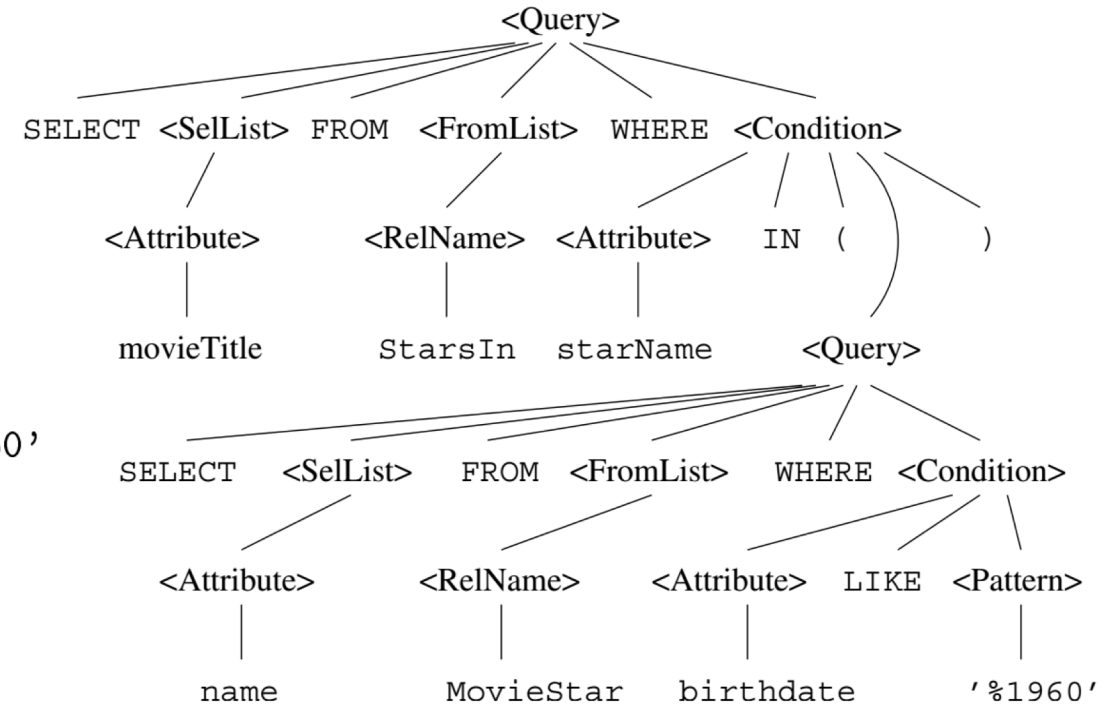


- Formal definition of the SQL Language
  - Grammar rules.
  - Parse diagrams/machines.
- You will learn about these concepts if you take:
  - COMS W3261: Computer Science Theory.
  - COMS W4115: Programming Languages and Translators.
- Parser/Translator:
  - Validates the syntax is correct.
  - Validates that the symbols are correct, e.g. the table exists.
  - Converts the statement into an intermediate form that can be compiled or interpreted, just like any other computer language.



SQL Parse Tree

```
SELECT movieTitle
FROM StarsIn
WHERE starName IN (
    SELECT name
    FROM MovieStar
    WHERE birthdate LIKE '%1960'
);
```



Simple Parsing Example

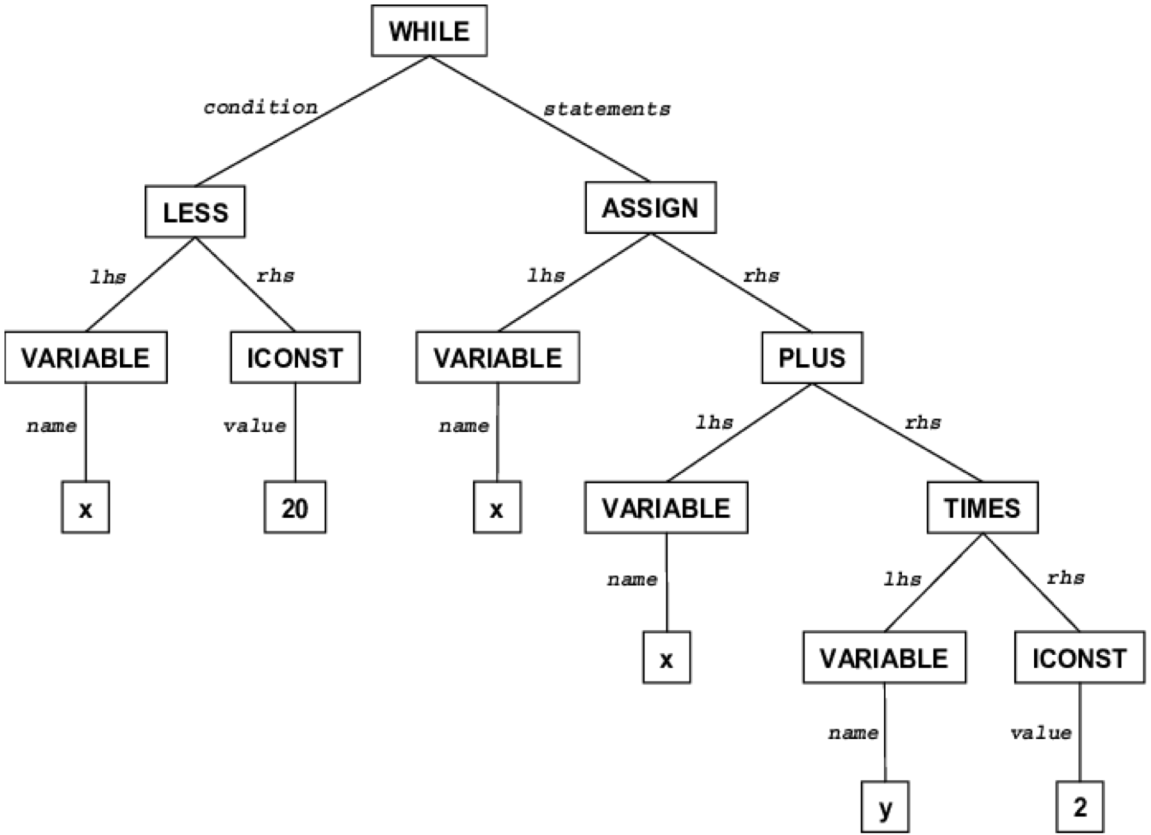
- https://github.com/mozilla/moz-sql-parser

```
from moz_sql_parser import parse
import json


q = """
select playerid, h as hits, ab as at_bats from
people join batting
on people.playerid=batting.playerid
"""

p = parse(q)
print("Parsed = ", json.dumps(p, indent=2))
```

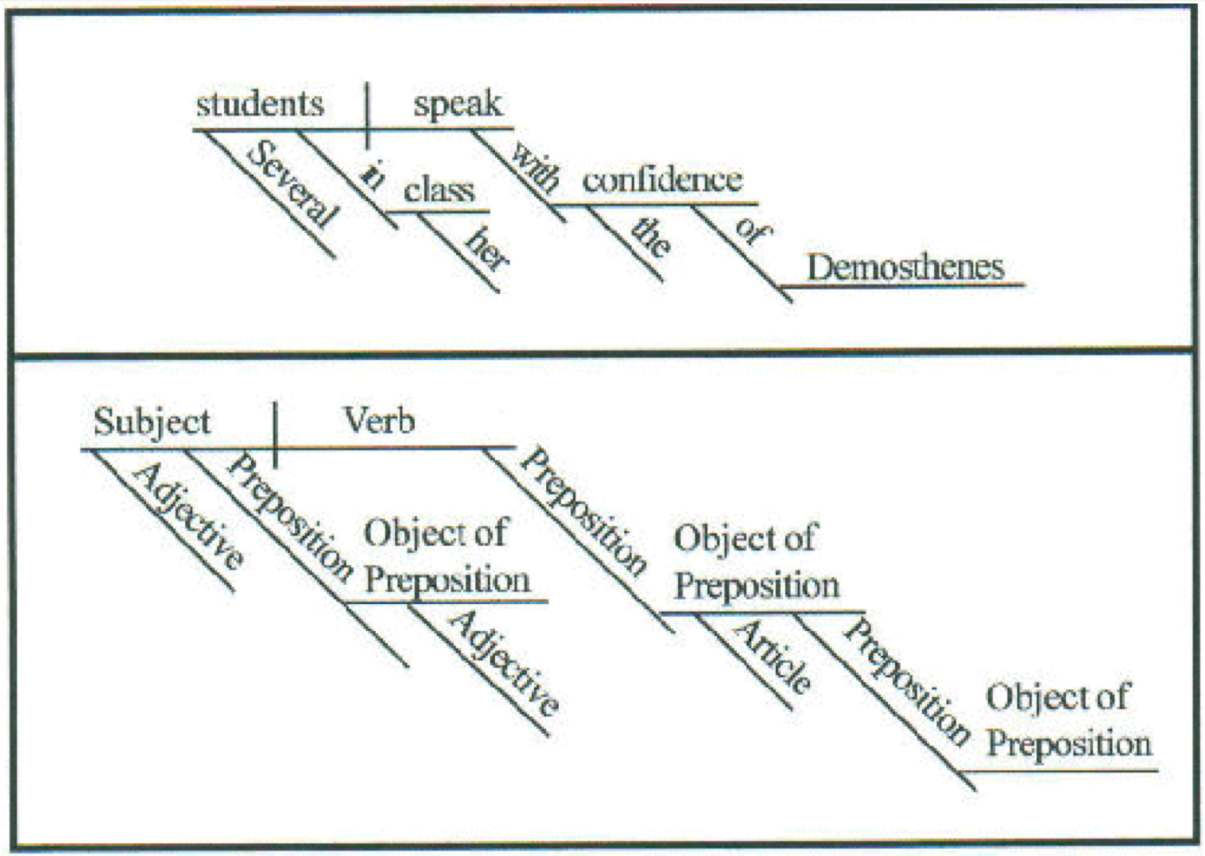
```
/Library/Frameworks/Python.framework/Versions/3.6/bir
Parsed =  {
  "select": [
    {
      "value": "playerid"
    },
    {
      "value": "h",
      "name": "hits"
    },
    {
      "value": "ab",
      "name": "at_bats"
    }
  ],
  "from": [
    "people",
    {
      "join": "batting",
      "on": {
        "eq": [
          "people.playerid",
          "batting.playerid"
        ]
      }
    }
  ]
}
```

Lecture: Skim code at
https://github.com/mozilla/moz-sql-parser

Python While Loop Syntax Tree



Common Concept – Example, Sentence Diagram



This is how we learned grammar in parochial school.

Instead of error messages when wrong, we got our knuckles smacked.

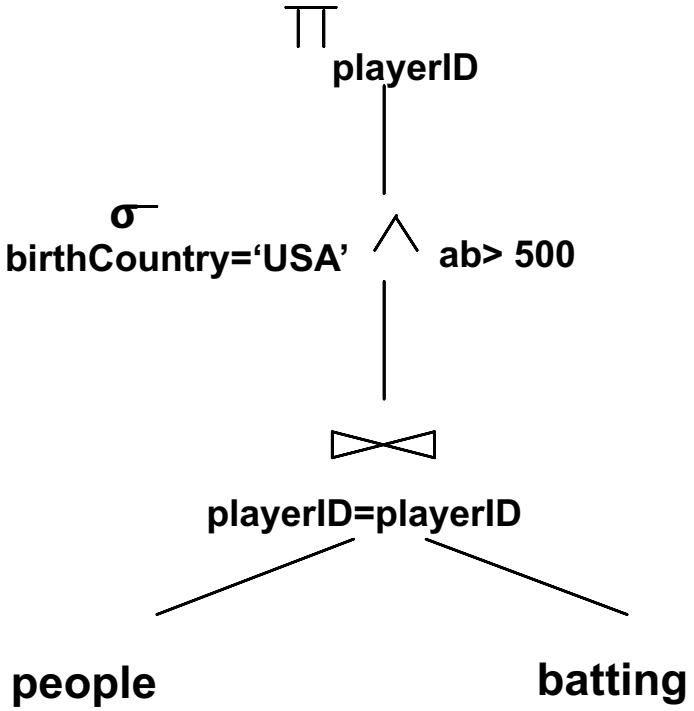
I have very good grammar skills

Query Plan/Logical Plan

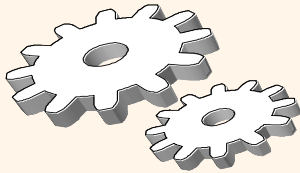
Simple Execution/Plan Tree

```
SELECT  P.playerID
FROM  people P, batting B
WHERE  P.playerID=B.playerID
    AND
    P.birthCountry='USA'  AND
    B.ab>500
```

- The nodes are SQL operators. The operator parameters are:
  - Relations.
  - Parameters to operators.
- The edges are the flow of data. The data flowing is a relation.
  - Base table.
  - Derived table.



Relational Operations



- We will consider how to implement:
  - *Selection* (σ) Selects a subset of rows from relation.
  - *Projection* (π) Deletes unwanted columns from relation.
  - *Join* ($\bowtie$) Allows us to combine two relations.
  - *Set-difference* ($-$) Tuples in reln. 1, but not in reln. 2.
  - *Union* ($\cup$) Tuples in reln. 1 and in reln. 2.
  - *Aggregation* (SUM, MIN, etc.) and GROUP BY
- Since each op returns a relation, ops can be *composed*! After we cover the operations, we will discuss how to *optimize* queries formed by composing them.

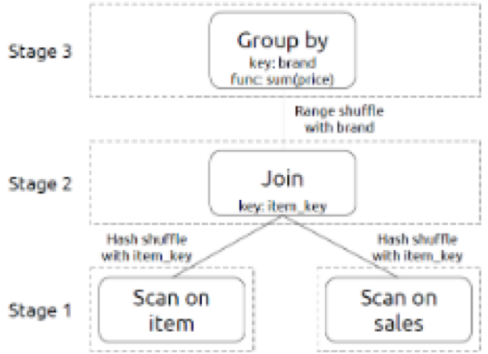
Logical Plan

SQL

```
SELECT
  item.brand,
  SUM(price)
FROM
  sales,
  item
WHERE
  sales.item_key =
  item.item_key
GROUP BY
  item.brand
```

Logical Query Plan

Distributed Query Plan (Master Worker)



https://gerardnico.com/data/type/relation/engine/query_plan

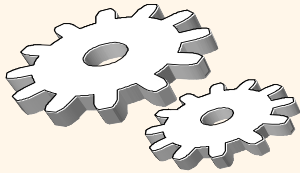
- The plan executes in stages (bottom up).
- Some stages can have parallelism, e.g. multiple threads to improve CPU utilization while processing I/O.

Homework 3 – Part IIa

```
def execute_join(self, left_r, right_r, on_fields, where_template, project_fields):
    """
    Implements a JOIN on two CSV Tables. Support equi-join only on a list of common
    columns names.
    :param left_r: The left table, or first input table
    :param right_r: The right table, or second input table.
    :param on_fields: A list of common fields used for the equi-join.
    :param where_template: Select template to apply to the result to determine what to return.
    :param project_fields: List of fields to return from the result.
    :return: List of dictionary elements, each representing a row.
    """
    pass
```

- We have developed a basic CSVDataTable supporting *find_by_template()*, *insert()*, and *delete()*.
- We will add support for execute_join()

Optimization/Execution Overview



Overview of Query Evaluation

- *Plan: Tree of R.A. ops, with choice of alg for each op.*
  - Each operator typically implemented using a `pull' interface: when an operator is `pulled' for the next output tuples, it `pulls' on its inputs and computes them.
- Two main issues in query optimization:
  - For a given query, what plans are considered?
    - Algorithm to search plan space for cheapest (estimated) plan.
  - How is the cost of a plan estimated?
- Ideally: Want to find best plan. Practically: Avoid worst plans!

Query Compilation

Database Systems: The Complete Book (2nd Edition) 2nd Edition
by Hector Garcia-Molina (Author), Jeffrey D. Ullman (Author), Jennifer Widom (Author)

Preview of Query Compilation

To set the context for query execution, we offer a very brief outline of the content of the next chapter. Query compilation is divided into the three major steps shown in Fig. 15.2.

a) *Parsing.* A *parse tree* for the query is constructed.

b) *Query Rewrite.* The parse tree is converted to an initial query plan, which is usually an algebraic representation of the query. This initial plan is then transformed into an equivalent plan that is expected to require less time to execute.

c) *Physical Plan Generation.* The abstract query plan from (b), often called a *logical query plan*, is turned into a *physical query plan* by selecting algorithms to implement each of the operators of the logical plan, and by selecting an order of execution for these operators. The physical plan, like the result of parsing and the logical plan, is represented by an expression tree. The physical plan also includes details such as how the queried relations are accessed, and when and if a relation should be sorted.

Query Optimization

Parts (b) and (c) are often called the *query optimizer*, and these are the hard parts of query compilation. To select the best query plan we need to decide:

1. Which of the algebraically equivalent forms of a query leads to the most efficient algorithm for answering the query?

2. For each operation of the selected form, what algorithm should we use to implement that operation?

3. How should the operations pass data from one to the other, e.g., in a pipelined fashion, in main-memory buffers, or via the disk?

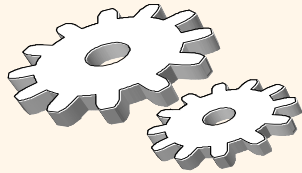
Each of these choices depends on the metadata about the database. Typical metadata that is available to the query optimizer includes: the size of each relation; statistics such as the approximate number and frequency of different values for an attribute; the existence of certain indexes; and the layout of data on disk.

Three Core Optimization Techniques

- Query rewrite: Transform the logical query into an equivalent, more efficient query (lower cost query), e.g.
  - R ⋈ S is the same as S ⋈ R
  - $\boldsymbol{\sigma}$(R ⋈ S) is the same at $\boldsymbol{\sigma}$(R) ⋈ $\boldsymbol{\sigma}$(S)
  - $\boldsymbol{\delta}$(R ⋈ S) = $\boldsymbol{\delta}$(R) ⋈ $\boldsymbol{\delta}$(S), where $\boldsymbol{\delta}$ is the distinct operator.
- Access path selection: Which index to choose?
- Operator implementation selection:
  - There are several related implementations for each operator
  - For example,
    - *Sort Scan*
    - *Sort Join*
    - *Hash Join*
    - *Hash Distinct*

Query execution cost

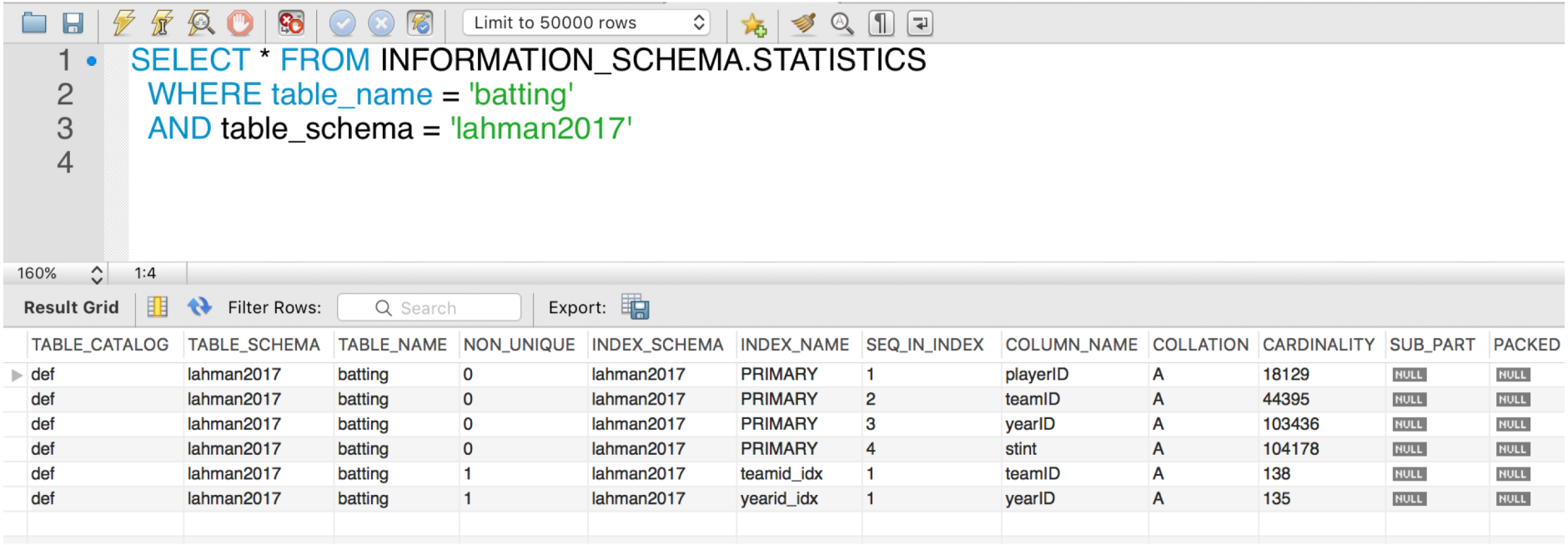
- Query execution cost is usually a weighted sum of the I/O cost (# disk accesses) and CPU cost (msec)
  - w * IO_COST + CPU_COST
- Basic Idea:
  - Cost of an operator depends on input data size, data distribution, physical layout
  - The optimizer uses statistics about the relations to *estimate* the cost
  - Need statistics on base relations and intermediate results



Statistics and Catalogs

- Need information about the relations and indexes involved. ***Catalogs*** typically contain at least:
  - # tuples (NTuples) and # pages (NPages) for each relation.
  - # distinct key values (NKeys) and NPages for each index.
  - Index height, low/high key values (Low/High) for each tree index.
- Catalogs updated periodically.
  - Updating whenever data changes is too expensive; lots of approximation anyway, so slight inconsistency ok.
- More detailed information (e.g., histograms of the values in some field) are sometimes stored.

MySQL Example



TABLE_CATALOG	TABLE_SCHEMA	TABLE_NAME	NON_UNIQUE	INDEX_SCHEMA	INDEX_NAME	SEQ_IN_INDEX	COLUMN_NAME	COLLATION	CARDINALITY	SUB_PART	PACKED
def	lahman2017	batting	0	lahman2017	PRIMARY	1	playerID	A	18129	NULL	NULL
def	lahman2017	batting	0	lahman2017	PRIMARY	2	teamID	A	44395	NULL	NULL
def	lahman2017	batting	0	lahman2017	PRIMARY	3	yearID	A	103436	NULL	NULL
def	lahman2017	batting	0	lahman2017	PRIMARY	4	stint	A	104178	NULL	NULL
def	lahman2017	batting	1	lahman2017	teamid_idx	1	teamID	A	138	NULL	NULL
def	lahman2017	batting	1	lahman2017	yearid_idx	1	yearID	A	135	NULL	NULL

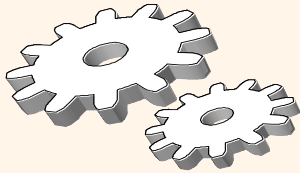
The Catalog Contains

- For each table
  - Table name, storage information.
  - Attribute name and type for each column.
  - Index name, type and indexed attributes
  - Constraints
- Statistical information
  - Cardinality of each table
  - Size: No. of blocks.
  - Index cardinality: Number of distinct key values for each index
  - Index size: Number of blocks for each index.
  - Index tree height
  - Index ranges

Access Path

- Every relational operator accepts one or more tables as input.
  The operators "accesses the tuples" in the tables.
- There are two "ways" to retrieve the tuples
  - Scan the relation (via blocks)
  - Use an index and matching condition.
- A selection condition is in *conjuctive normal form* if
  - Each term is column_name op value
  - op is one of <, <=, =, >, >=, <>
  - The terms are combined, e.g. (nameLast = 'Ferguson') AND (ab > 500)
  - CNF allows using a matching index for the access path.
- The engine can select a *matching index*
  - A Hash Index on (c1,c2,c3) if the condition is c1=x AND c2=y AND c3=z.
  - A Tree Index if the condition is (can be ordered as)
    - c1 op value
    - c1 op x AND/OR c2 op y
    - … …
  - Many indexes may match, and the engine chooses the *most selective match* (number of tuples or blocks) that match.

Access Paths



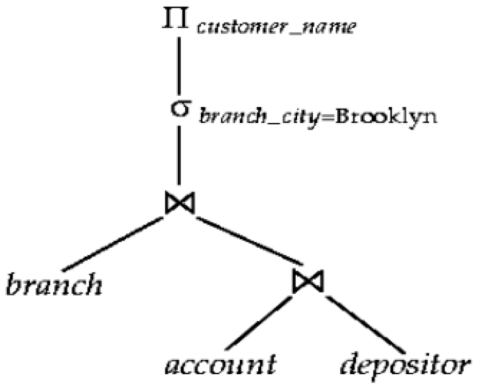
- An access path is a method of retrieving tuples:
  - File scan, or index that matches a selection (in the query)
- A tree index *matches* (a conjunction of) terms that involve only attributes in a *prefix* of the search key.
  - E.g., Tree index on <*a, b, c*> matches the selection *a=5 AND b=3*, and *a=5 AND b>6*, but not *b=3*.
- A hash index *matches* (a conjunction of) terms that has a term *attribute = value* for every attribute in the search key of the index.
  - E.g., Hash index on <*a, b, c*> matches *a=5 AND b=3 AND c=5*; but it does not match *b=3, or a=5 AND b=3, or a>5 AND b=3 AND c=5*.

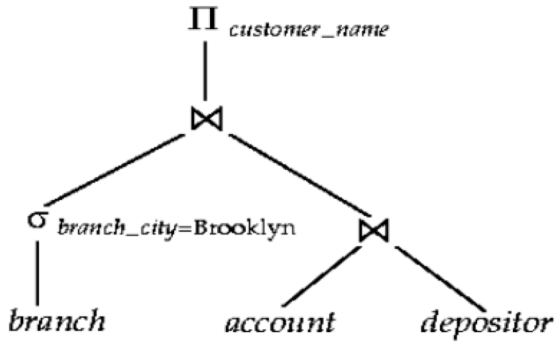
Query Rewrite

Many Ways to Evaluate a Query

Query evaluation

- Alternative ways of evaluating a given query
  - Equivalent expressions
  - Different algorithms for each operation





Simple Technique – SELECTion Pushing

- Consider two relations
  - StarsIn(title, year, star_name)
  - Movies(title, year, length, genre, studio_name, producer_no)
- Find everyone who starred in a comedy movie in 1996
  - Query 1:

```
SELECT star_name,  FROM StarsIn JOIN Movies ON
   StarsIn.title=Movies.title AND StarsIn.year=Movies.year
   WHERE StarsIn.year=1996 and Movies.genre='Comedy'
```

  - Query 2:

```
SELECT star_name FROM
   (SELECT star_name, title, year FROM StarsIn WHERE year=1996 as a) JOIN
   (SELECT title, year, genre FROM Movies WHERE year=1996 AND genre='Comedy')
   ON a.year=b.year and a.title=b.title
```

These queries are equivalent, but (2) is much more efficient.

Simple Technique – SELECTion Pushing

- SELECT on a JOIN of table R and table S compares O(#(R)*#(S)) tuples
  - Have to load relevant blocks
  - Compare the tuples to compute the JOIN
  - Scan the JOIN to apply the SELECT
- Pushing the SELECT(s) through the JOINs
  - May vastly reduce the size of the table to JOIN
  - Only JOIN tuples that could be selected in the WHERE clause.

You can also push project through JOIN, but this reduces the size of the data, not the number of tuples examined.

Cannonical Example – SELECT Pushdown

- The basic rule: $\boldsymbol{\sigma}(R \bowtie S) = \boldsymbol{\sigma}(R) \bowtie \boldsymbol{\sigma}(S)$
- There are several ways this rewrite can help:
  - Complexity goes from #(R) * #(S) to #($\boldsymbol{\sigma}$(R)) * #($\boldsymbol{\sigma}$(S)) [Note: #(X) is no. of tuples in X]
  - I/O (let B(X) be the number of disk blocks for X):
    - I scan R either way, which means I read EVERY disk block one time, B(R)
    - If I can allocate N buffer frames to the probe table, the probability of a cache hit goes from N/[#(S)] to N/[#($\boldsymbol{\sigma}$(S))], and I may be able to hold the entire $\boldsymbol{\sigma}$(S) in memory.
  - The JOINed table is derived, and does not have indexes. The engine may be able to use indexes for $\boldsymbol{\sigma}$(R) and $\boldsymbol{\sigma}$(S) The basic rule: $\boldsymbol{\sigma}(R \bowtie S) = \boldsymbol{\sigma}(R) \bowtie \boldsymbol{\sigma}(S)$
- The rule $\pi(R \bowtie S) = \pi(R) \bowtie \pi(S)$ may have I/O benefits for probing S. The query execution can cache more of the tuples in buffer frames.
- NOTE: This is an example of query rewrite and *two-pass algorithms*.

Swapping Probe and Scan Tables

```
def define_tables():
    """
    This would be done by a DBA. This code is simulating CREATE TABLE statements.
    The underlying CSV file contains the data. HW3 assumes some other code performs insert, update, delete
    on the data files.
    :return: None
    """
    cleanup()
    cat = CSVCatalog.CSVCatalog()                                    # Connect to catalog to perform DDL

    # Define columns.
    cds = []
    cds.append(CSVCatalog.ColumnDefinition("playerID", "text", True))
    cds.append(CSVCatalog.ColumnDefinition("nameLast", "text", True))
    cds.append(CSVCatalog.ColumnDefinition("nameFirst", column_type="text"))
    cds.append(CSVCatalog.ColumnDefinition("birthCity", "text"))
    cds.append(CSVCatalog.ColumnDefinition("birthCountry", "text"))
    cds.append(CSVCatalog.ColumnDefinition("throws", column_type="text"))

    # CREATE TABLE DDL. Put definition information in our version of INFORMATION_SCHEMA.
    t = cat.create_table("people",
                         "/Users/donaldferguson/Dropbox/ColumbiaCourse/Courses/Fall2018/W4111/Data/People.csv", cds)
    t.define_index("pid_idx", "INDEX", ['playerID'])

    cds = []
    cds.append(CSVCatalog.ColumnDefinition("playerID", "text", True))
    cds.append(CSVCatalog.ColumnDefinition("H", "number", True))
    cds.append(CSVCatalog.ColumnDefinition("AB", column_type="number"))
    cds.append(CSVCatalog.ColumnDefinition("teamID", "text", True))
    cds.append(CSVCatalog.ColumnDefinition("yearID", "text", True))
    cds.append(CSVCatalog.ColumnDefinition("stint", column_type="number", not_null=True))

    t = cat.create_table("batting",
                         "/Users/donaldferguson/Dropbox/ColumbiaCourse/Courses/Fall2018/W4111/Data/Batting.csv", cds)
```

- This index does not help for R ⋈ S if the JOIN column is playerID.
  - I have to look at every row in R.
  - To form probe in S with the current row's playerID.
- But, R ⋈ S is the same as S ⋈ R
- If I
  - Scan S to form the probe query into R on playerID,
  - I can probe via the index.

Swapping Probe and Scan Tables

```
def test_join_4():
    people_tbl = CSVTable.CSVTable("people")
    batting_tbl = CSVTable.CSVTable("batting")
    start_time = time.time()
    result = people_tbl.join(batting_tbl, on_fields=['playerID'])
    end_time = time.time()
    print("\nElapsed time to execute join = ", end_time - start_time)
    print("Result = \n", result)
```

```
/Library/Frameworks/Python.framework/Versions/3.6/bin/python3.6 /Users/donaldferguson/Dropbo
Swapping scan and probe tables.
Before pushdown, scan table size is =  102816
Attempting to pushdown WHERE template =  null
After pushdown, scan table size is =  102816

Elapsed time to execute join =  1.089634656906128
Result =
 Name: JOIN(people,batting) File: DERIVED
Row count: 102815


Sample rows:
AB          H           playerID    stint       teamID      yearID      bi
4           0           abercda01   1           TRO         1871        Fo
118         32          addybo01    1           RC1         1871        Po
137         40          allisar01   1           CL1         1871        Ph
133         44          allisdo01   1           WS3         1871        Ph
120         39          ansonca01   1           RC1         1871        Ma
...         ...         ...         ...         ...         ...         ..
0           0           zychto01    1           SEA         2016        Mo
164         34          zuninmi01   1           SEA         2016        Ca
523         142         zobribe01   1           CHN         2016        Eu
427         93          zimmery01   1           WAS         2016        Wa
4           1           zimmejo02   1           DET         2016        Au
```

- The JOIN algorithm
  - Detects an index that applies to ON playerID
  - Swaps the scan and probe tables.
- There is no WHERE clause to pushdown.
- But,
  - I have a hash index on people.PlayerID
  - Query cost goes
    - From #(R) * #(S)
    - To #(S) * c, where c is a constant.
    - Would be #(S) * log[#(R)] for B+ Tree
- Also not,
  - Because of the algorithm being a nested loop,
  - The row order changed in the result.

Putting the Pieces Together

```
def test_join_5():
    people_tbl = CSVTable.CSVTable("people")
    batting_tbl = CSVTable.CSVTable("batting")
    start_time = time.time()
    result = people_tbl.join(batting_tbl, on_fields=['playerID'],
        where_template={"nameLast": "Williams", "teamID": "BOS"})
    end_time = time.time()
    print("\nElapsed time to execute join = ", end_time - start_time)
    print("Result = \n", result)
```

- Detects index for ON clause and swaps scan and probe tables.
- Pushes down WHERE clause onto scan table.

```
Swapping scan and probe tables.
Before pushdown, scan table size is =  102816
Attempting to pushdown WHERE template =  {"nameLast": "Williams", "teamID": "BOS"}
After pushdown, scan table size is =  4328

Elapsed time to execute join =  0.0972909927368164
Result =
 Name: JOIN(people,batting) File: DERIVED
Row count: 32


Sample rows:
```

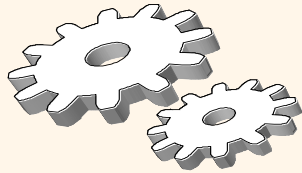
AB	H	playerID	stint	teamID	yearID	birthCity	birthCountry	nameFirst	nameLast	throws
9	3	willida02	1	BOS	1902	Scranton	USA	Dave	Williams	L
284	68	williri01	1	BOS	1911	Carthage	USA	Rip	Williams	R
85	31	willide01	1	BOS	1924	Portland	USA	Denny	Williams	R
218	50	willide01	1	BOS	1925	Portland	USA	Denny	Williams	R
18	4	willide01	1	BOS	1928	Portland	USA	Denny	Williams	R
...	...	...	...	...	...	...	...	...	...	...
0	0	willira01	1	BOS	2011	Harlingen	USA	Randy	Williams	L
5	1	willida06	1	BOS	1989	Weirton	USA	Dana	Williams	R
0	0	willist02	1	BOS	1972	Enfield	USA	Stan	Williams	R
69	11	willidi02	1	BOS	1964	St. Louis	USA	Dick	Williams	R
136	35	willidi02	1	BOS	1963	St. Louis	USA	Dick	Williams	R

Algorithms

Hash JOIN

- Consider SELECT * FROM people JOIN batting on people.playerID = batting.playerID
- Assume:
  - People has 2 data blocks.
  - Batting has 2 data blocks.
  - I can allocate 3 buffer frames to the JOIN
    - One block holds the current block of R for the scan.
    - One holds a block of S.
    - One holds the JOIN result block I am current constructing.
  - For each row in R, the probability of a buffer hit is 1/2.
  - This means that I will do (1/2) * #(R) I/Os to probe S.
- An alternate approach is to use a hash value on playerID to
  - Read R and partition into 2 buckets, each of size one block.
  - Read S and partition into 2 buckets, each of size one block.
  - I can process the join by loading the 1st block of the hashed R and 1st block of the hashed S into the buffer pool.
  - This means the S tuples matching the current playerID are in the buffer pool.
  - I do some extra upfront I/Os to read, partition and write the hashed tables.
  - But the nested loop is much, much more efficient.
- We still have to scan the "probe" blocks when in memory, but we can hash again.

A Note on Complex Selections



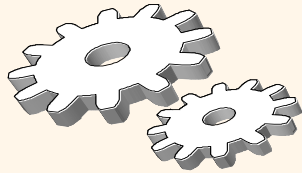
(day<8/9/94 AND rname='Paul') OR bid=5 OR sid=3

- Selection conditions are first converted to *conjunctive normal form* (CNF):

  (day<8/9/94 OR bid=5 OR sid=3) AND
  (rname='Paul' OR bid=5 OR sid=3)

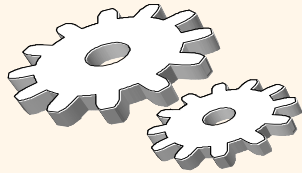
- We only discuss case with no ORs; see text if you are curious about the general case.

One Approach to Selections



- Find the *most selective access path*, retrieve tuples using it, and apply any remaining terms that don't match the index:
  - *Most selective access path:* An index or file scan that we estimate will require the fewest page I/Os.
  - Terms that match this index reduce the number of tuples *retrieved*; other terms are used to discard some retrieved tuples, but do not affect number of tuples/pages fetched.
  - Consider *day<8/9/94 AND bid=5 AND sid=3*. A B+ tree index on *day* can be used; then, *bid=5* and *sid=3* must be checked for each retrieved tuple. Similarly, a hash index on <*bid, sid*> could be used; *day<8/9/94* must then be checked.

Using an Index for Selections

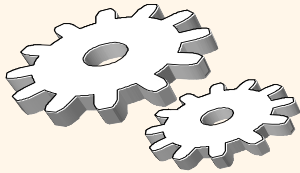


- Cost depends on #qualifying tuples, and clustering.
  - Cost of finding qualifying data entries (typically small) plus cost of retrieving records (could be large w/o clustering).
  - In example, assuming uniform distribution of names, about 10% of tuples qualify (100 pages, 10000 tuples). With a clustered index, cost is little more than 100 I/Os; if unclustered, upto 10000 I/Os!

```
SELECT  *
FROM    Reserves R
WHERE   R.rname < 'C%'
```

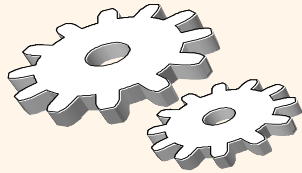
Projection

```
SELECT   DISTINCT
         R.sid, R.bid
FROM     Reserves R
```



- The expensive part is removing duplicates.
  - SQL systems don't remove duplicates unless the keyword DISTINCT is specified in a query.
- Sorting Approach: Sort on <sid, bid> and remove duplicates. (Can optimize this by dropping unwanted information while sorting.)
- Hashing Approach: Hash on <sid, bid> to create partitions. Load partitions into memory one at a time, build in-memory hash structure, and eliminate duplicates.
- If there is an index with both R.sid and R.bid in the search key, may be cheaper to sort data entries!

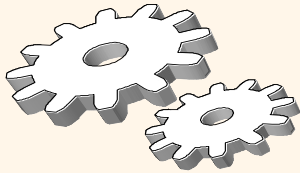
Join: Index Nested Loops



foreach tuple r in R do
    foreach tuple s in S where r_i == s_j do
        add <r, s> to result

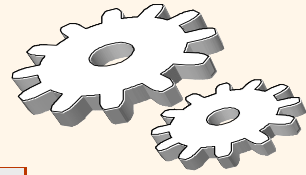
- If there is an index on the join column of one relation (say S), can make it the inner and exploit the index.
  - Cost: M + ((M*p_R) * cost of finding matching S tuples)
  - M=#pages of R, p_R=# R tuples per page
- For each R tuple, cost of probing S index is about 1.2 for hash index, 2-4 for B+ tree. Cost of then finding S tuples (assuming Alt. (2) or (3) for data entries) depends on clustering.
  - Clustered index: 1 I/O (typical), unclustered: upto 1 I/O per matching S tuple.

Join: Sort-Merge ($R \bowtie_{i=j} S$)



- Sort R and S on the join column, then scan them to do a ``merge'' (on join col.), and output result tuples.
  - Advance scan of R until current R-tuple >= current S tuple, then advance scan of S until current S-tuple >= current R tuple; do this until current R tuple = current S tuple.
  - At this point, all R tuples with same value in Ri (*current R group*) and all S tuples with same value in Sj (*current S group*) *match*; output <r, s> for all pairs of such tuples.
  - Then resume scanning R and S.
- R is scanned once; each S group is scanned once per matching R tuple. (Multiple scans of an S group are likely to find needed pages in buffer.)

Example of Sort-Merge Join



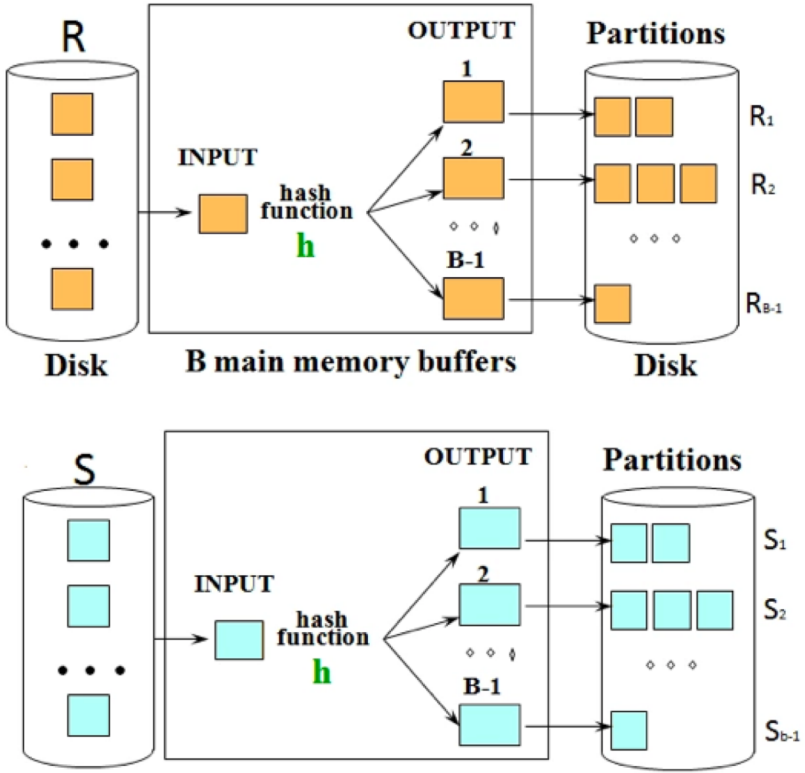
sid	sname	rating	age
22	dustin	7	45.0
28	yuppy	9	35.0
31	lubber	8	55.5
44	guppy	5	35.0
58	rusty	10	35.0

sid	bid	day	rname
28	103	12/4/96	guppy
28	103	11/3/96	yuppy
31	101	10/10/96	dustin
31	102	10/12/96	lubber
31	101	10/11/96	lubber
58	103	11/12/96	dustin

- Cost: M log M + N log N + (M+N)
  - The cost of scanning, M+N, could be M*N (very unlikely!)
- With 35, 100 or 300 buffer pages, both Reserves and Sailors can be sorted in 2 passes; total join cost: 7500.

Hash JOIN (http://cs186.wikia.com/wiki/Joins)

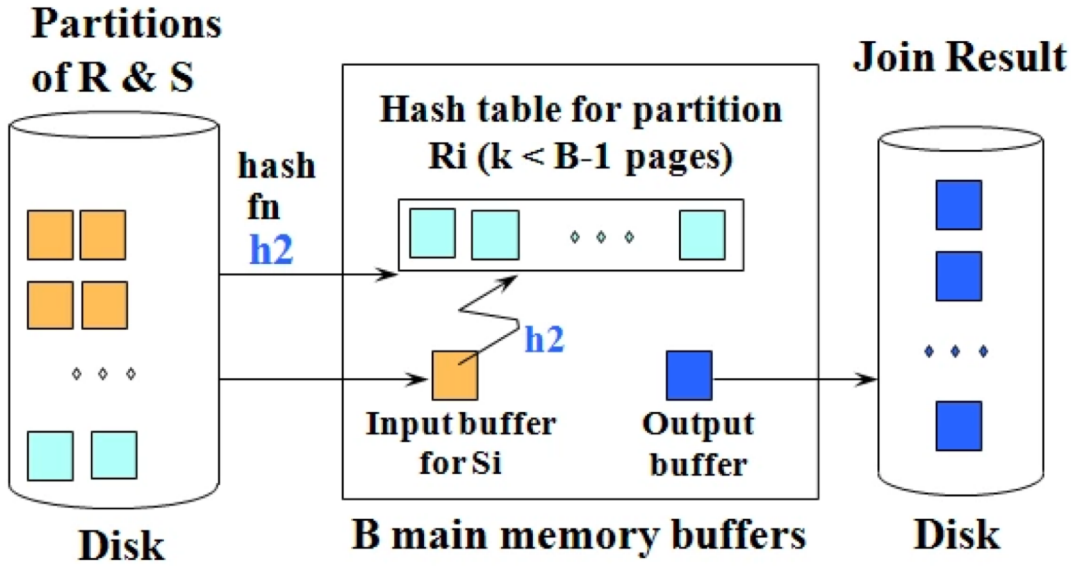
- Hash join uses two stages to accomplish the join.
  - The initial **partitioning** phase,
  - Followed by the **probing** phase.
- In the first stage, we hash the two relations into partitions on disk using a hash function that partitions based on the join condition. The relation *R* gets partitioned into *Ri* and the relation *S* gets partitioned into *Si*. The diagram on the right demonstrates the first stage of the hash join. We will see that in the second stage, we will match *Ri* to *Si* partitions.



Hash JOIN (http://cs186.wikia.com/wiki/Joins)

In the second stage,

- We use an in-memory hash table to perform the joining.
- Read in partition *Ri* and hash it into the in memory hash table
- Scan partition *Si* and probe the in memory hash table for matches.
- Matches go to the output buffer



Homework 3 – Discussion