Carnegie Mellon University

Sorting & Joins

Lecture #11

Database Systems
15-445/15-645
Fall 2017

Andy Pavlo
Computer Science Dept.
Carnegie Mellon Univ.

ADMINISTRIVIA

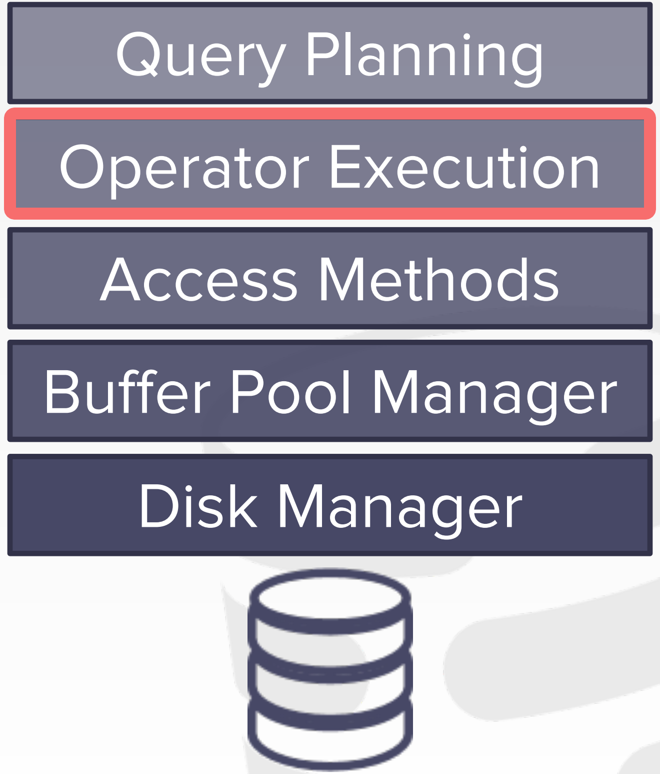
Homework #3 is due TODAY @ 11:59pm

Homework #4 is due Wednesday October 11th @ 11:59pm

STATUS

We will continue our discussion on how the DBMS executes queries.

We will focus on a couple of frequently used relational operators.



TODAY'S AGENDA

Sorting algorithms
Join algorithms

WHY DO WE NEED TO SORT?

Relational model

→ Tuples in a table have no specific order

SELECT...ORDER BY

→ Users often want to retrieve tuples in a specific order
→ Trivial to support duplicate elimination (DISTINCT)
→ Bulk loading sorted tuples into a B+ tree index is faster

SELECT...GROUP BY

→ Sort-merge join algorithm

SORTING ALGORITHMS

Data fits in memory: Then we can use a standard sorting algorithm like **quick-sort**.

Data does not fit in memory: Sorting data that does not fit in main-memory is called **external sorting**.

EXTERNAL MERGE SORT

A frequently used external sorting algorithm.

Idea: Hybrid **sort-merge** strategy

→ Sorting phase: Sort small chunks of data that fit in main-memory, and then write back the sorted data to a file on disk.

→ Merge phase: Combine sorted sub-files into a single larger file.

OVERVIEW

Let’s start with a simple example:

2-way external merge sort.

Later generalize it to **k-way** external merge sort.

Files are broken up into N pages.

The DBMS has a finite number of B fixed-size buffers.

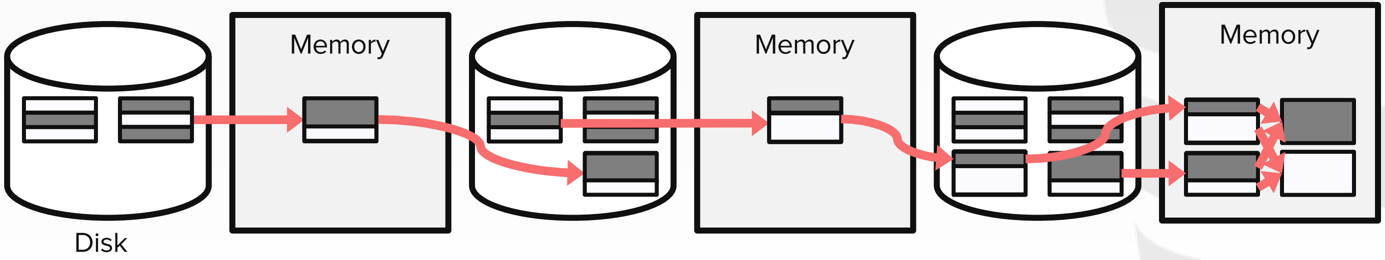
2-WAY EXTERNAL MERGE SORT

Pass 0:

→ Reads every B pages of the table into memory
→ Sorts them, and writes them back to disk.
→ Each sorted set of pages is a **run**

Pass 1,2,3,...:

→ Recursively merges pairs of runs into runs twice as long
→ Uses three buffer pages (two for input pages, one for output)



2-WAY EXTERNAL MERGE SORT

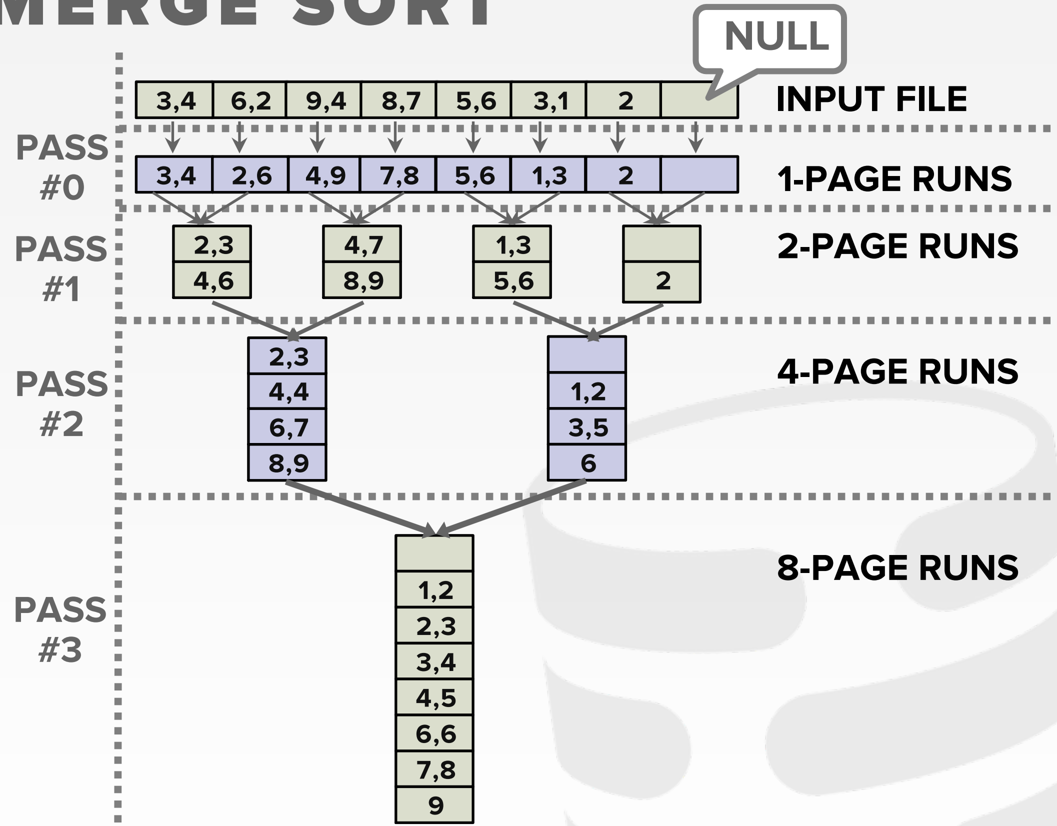
In each pass, we read and write each page in file.

Number of passes
$= 1 + \lceil \log_2 N \rceil$

Total I/O cost
$= 2N \cdot (\text{\# of passes})$

Divide and conquer strategy:

Sort sub-files and merge



2-WAY EXTERNAL MERGE SORT

This algorithm only requires **three** buffer pages (B=3).

Even if we have more buffer space available (B>3), it does not effectively utilize them.

Let's next generalize the algorithm to make use of extra buffer space.

GENERAL EXTERNAL MERGE SORT

Pass 0: Use B buffer pages.

Produce $\lceil N / B \rceil$ sorted runs of size B

Pass 1,2,3,…: Merge B-1 runs. (**K-way merge**)

Number of passes $= 1 + \lceil \log_{B-1} \lceil N / B \rceil \rceil$

Total I/O Cost $= 2N \cdot (\text{\# of passes})$

K-WAY MERGE ALGORITHM

Input: K sorted sub-arrays

Efficiently computes the **minimum element** of all K sub-arrays

Repeatedly transfers that element to output array

Internally maintains a heap to efficiently compute minimum element

Time Complexity = $O(N \log_2 K)$

EXAMPLE

Sort 108 page file with 5 buffer pages: N=108, B=5

→ Pass 0: $\lceil N / B \rceil = \lceil 108 / 5 \rceil = 22$ sorted runs of 5 pages each (last run is only 3 pages)

→ Pass 1: $\lceil N' / B\text{-}1 \rceil = \lceil 22 / 4 \rceil = 6$ sorted runs of 20 pages each (last run is only 8 pages)

→ Pass 2: $\lceil N'' / B\text{-}1 \rceil = \lceil 6 / 4 \rceil = 2$ sorted runs, 80 pages and 28 pages

→ Pass 3: Sorted file of 108 pages

$$1+\lceil \log_{B-1}\lceil N / B \rceil \rceil = 1+\lceil \log_4 22 \rceil = 1+\lceil 2.229... \rceil \rightarrow \underline{4 \text{ passes}}$$

USING B+TREES

Scenario: Table that must be sorted already has a B+ tree index on the sort attribute(s).

Can we accelerate sorting?

Idea: Retrieve tuples in desired sort order by simply traversing the leaf pages of the tree.

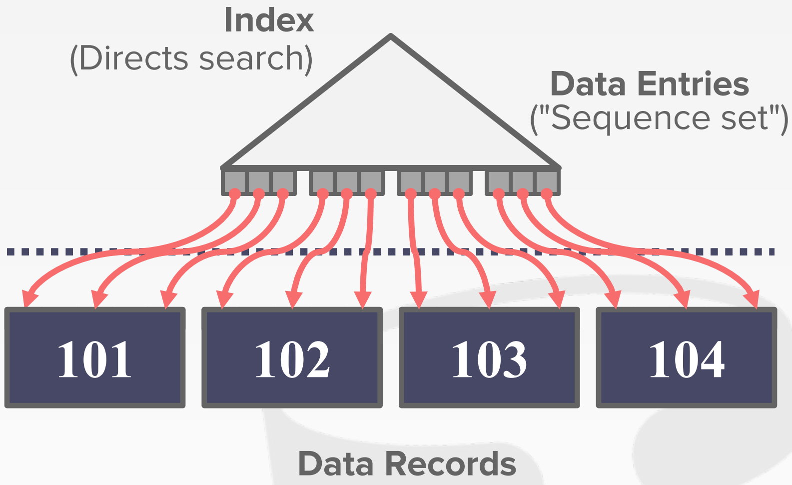
Cases to consider:

→ Clustered B+ tree
→ Unclustered B+ tree

CASE 1: CLUSTERED B+TREE

Traverse to the left-most leaf page, and then retrieve tuples from all leaf pages.

Always better than external sorting. Good idea!

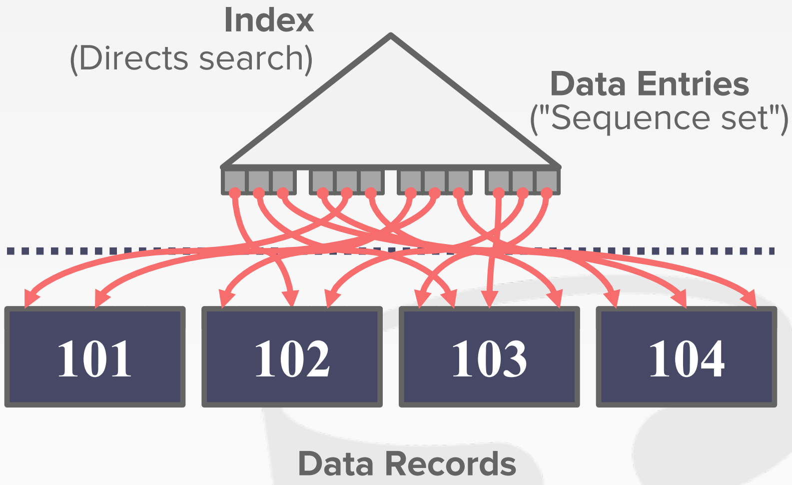


CARNEGIE MELLON DATABASE GROUP

CASE 2: UNCLUSTERED B+TREE

Chase each pointer to the page that contains the data.

In general, one I/O per data record. Bad idea!!



ALTERNATIVES TO SORTING

What if we don't need the data to be ordered?

→ Forming groups in GROUP BY (no ordering)

→ Removing duplicates in DISTINCT (no ordering)

Can we remove duplicates without sorting?

→ Hashing is a better alternative in this scenario

→ Only need to remove duplicates, no need for ordering

→ Can be computationally cheaper than sorting!

SORTING: SUMMARY

External merge sort minimizes disk I/O

→ Pass 0: Produces sorted runs of size B

→ Later Passes: Recursively merge runs

Next week: Query optimizer picks a **sorting or hashing operator** based on ordering requirements in the query plan.

CARNEGIE MELLON DATABASE GROUP

TODAY'S AGENDA

Sorting algorithms

Join algorithms

WHY DO WE NEED TO JOIN?

Relational model
→ Unnecessary repetition of information must be avoided
→ We decompose tables using normalization theory

SELECT...JOIN
→ Reconstruct original tables via joins
→ No information loss

Anybody here into sailing?

Hoofer Sailing Club

SAILING CLUB DATABASE

SAILORS

SID	SNAME	RATING	AGE
1	Andy	999	45.0
3	Obama	50	52.0
2	Tupac	32	26.0
6	Bieber	10	19.0

RESERVES

SID	BID	DAY	RNAME
6	103	2014-02-01	Matlock
1	102	2014-02-02	Macgyver
2	101	2014-02-02	A-team
1	101	2014-02-01	Dallas

Sailors(sid: int, sname: varchar, rating: int, age: real)
Reserves(sid: int, bid: int, day: date, rname: varchar)

SAILING CLUB DATABASE

SAILORS

SID	SNAME	RATING	AGE
1	Andy	999	45.0
3	Obama	50	52.0
2	Tupac	32	26.0
6	Bieber	10	19.0

Each tuple is 50 bytes
80 tuples per page
500 pages total
N=500, p_S=80

RESERVES

SID	BID	DAY	RNAME
6	103	2014-02-01	Matlock
1	102	2014-02-02	Macgyver
2	101	2014-02-02	A-team
1	101	2014-02-01	Dallas

Each tuple is 40 bytes
100 tuples per page
1000 pages total
M=1000, p_R=100

JOIN VS CROSS-PRODUCT

R⋈S is very common and thus must be carefully optimized.

R×S followed by a selection is inefficient because the cross-product is large.

There are many algorithms for reducing join cost, but no particular algorithm works well in all scenarios.

JOIN ALGORITHMS

Join algorithms we will cover in today's lecture:

→ Simple Nested Loop Join
→ Block Nested Loop Join
→ Index Nested Loop Join
→ Sort-Merge Join
→ Hash Join (next lecture)

I/O COST ANALYSIS

Assume:

→ M pages in R, p_R tuples per page, m tuples total

→ N pages in S, p_S tuples per page, n tuples total

→ In our examples, R is Reserves and S is Sailors.

We will consider more complex join conditions later.

Cost metric: # of I/Os

We will ignore output costs

JOIN QUERY EXAMPLE

```
SELECT *
  FROM Reserves R, Sailors S
  WHERE R.sid = S.sid
```

Assume that we don't know anything about the tables and we **don't** have any indexes.

JOIN ALGORITHMS

Join algorithms we will cover:

→ Simple Nested Loop Join
→ Block Nested Loop Join
→ Index Nested Loop Join
→ Sort-Merge Join

SIMPLE NESTED LOOP JOIN


```
foreach tuple r of R
  foreach tuple s of S
    output, if r and s match
```

SIMPLE NESTED LOOP JOIN

Outer table

```
foreach tuple r of R
  foreach tuple s of S
    output, if r and s match
```

Inner table

R(A,..)

S(A, ...)

SIMPLE NESTED LOOP JOIN

Why is this algorithm bad?

→ For **every tuple** in R, it scans S once

Number of disk accesses

→ Cost: $M + (m \cdot N)$

SIMPLE NESTED LOOP JOIN

Actual number:

→ $M + (m \cdot N) = 1000 + (100 \cdot 1000) \cdot 500 \approx 50$ M I/Os

→ At 0.1 ms/IO, Total time ≈ 1.3 hours

What if smaller table (S) is used as the outer table?

→ $N + (n \cdot M) = 500 + (80 \cdot 500) \cdot 1000 \approx 40$ M I/Os

→ Slightly better.

What assumptions are being made here?

→ 2 buffers for streaming the tables (and 1 for storing output)

JOIN ALGORITHMS

Join algorithms we will cover:

- → Simple Nested Loop Join
- → Block Nested Loop Join
- → Index Nested Loop Join
- → Sort-Merge Join

BLOCK NESTED LOOP JOIN

```
read block from R
  read block from S
    output, if a pair of tuples match
```

BLOCK NESTED LOOP JOIN

This algorithm performs fewer disk accesses.
→ For **every block** in R, it scans S once

Number of disk accesses
→ Cost: M + (M·N)

BLOCK NESTED LOOP JOIN

Algorithm Optimizations:

Which one should be the outer table?

→ The smaller table (in terms of # of pages)

BLOCK NESTED LOOP JOIN

Actual number:

→ M + (M·N) = 1000 + (1000 · 500) ≈ 0.5 M I/Os
→ At 0.1 ms/IO, Total time ≈ 50 seconds

What if we have B buffers available?

→ Use B-2 buffers for scanning outer table,
→ Use 1 buffer to scanning inner table, 1 buffer for storing output

BLOCK NESTED LOOP JOIN

```
read B-2 blocks from R
  read block from S
    output, if a pair of tuples match
```

BLOCK NESTED LOOP JOIN

This algorithm uses B-2 buffers for scanning M.

Number of disk accesses

→ Cost: $M + (\lceil M/(B-2) \rceil \cdot N)$

CARNEGIE MELLON DATABASE GROUP

BLOCK NESTED LOOP JOIN

What if the outer relation completely fits in memory (B>M+2)?

→ Cost: M + N = 1000 + 500 = 1500 I/Os

→ At 0.1ms/IO, Total time ≈ 0.15 seconds

JOIN ALGORITHMS

Join algorithms we will cover:

→ Simple Nested Loop Join
→ Block Nested Loop Join
→ Index Nested Loop Join
→ Sort-Merge Join

INDEX NESTED LOOP JOIN

Why do basic nested loop joins suck?

→ For each tuple in the outer table, we have to do a sequential scan to check for a match in the inner table.

Can we accelerate the join using an index?

Use an index to find **inner table matches**.

→ We could use an existing index for the join.

→ Or even build one on the fly.

INDEX NESTED LOOP JOIN

```
foreach tuple r of R
  foreach tuple s of S, where r_i = s_j
    output, if r_i and s_j match
```

Index Probe

CARNEGIE MELLON
DATABASE GROUP

INDEX NESTED LOOP JOIN

Number of disk accesses
→ Cost: M + (m · C)

Index Look-up Cost

NESTED LOOP JOIN: SUMMARY

Pick the smaller table as the outer table.

Buffer as much of the outer table in memory as possible.

Loop over the inner table or use an index.

JOIN ALGORITHMS

Join algorithms we will cover:
→ Simple Nested Loop Join
→ Block Nested Loop Join
→ Index Nested Loop Join
→ Sort-Merge Join

SORT-MERGE JOIN

Sort Phase: First sort **both** tables on the join attribute.

Merge Phase: Then scan the two sorted tables in parallel, and emit matching tuples.

WHEN IS SORT-MERGE JOIN USEFUL?

This join algorithm is useful if:

→ One or both tables are **already** sorted on join key

→ Output must be sorted on join key

Sorting: Might be achieved either by an explicit sort step, or by scanning the relation using an index on the join key.

SORT-MERGE JOIN EXAMPLE

```
SELECT *
  FROM Reserves R, Sailors S
  WHERE R.sid = S.sid
```

SID	SNAME	RATING	AGE
1	Andy	999	45.0
2	Tupac	32	26.0
3	Obama	50	52.0
6	Bieber	10	19.0

Sort!

SID	BID	DAY	RNAME
1	102	2014-02-02	Macgyver
1	101	2014-02-01	Dallas
2	101	2014-02-02	A-team
6	103	2014-02-01	Matlock

Sort!

SORT-MERGE JOIN EXAMPLE

```
SELECT *
  FROM Reserves R, Sailors S
  WHERE R.sid = S.sid
```

SID	SNAME	RATING	AGE
1	Andy	999	45.0
2	Tupac	32	26.0
3	Obama	50	52.0
6	Bieber	10	19.0

SID	BID	DAY	RNAME
1	102	2014-02-02	Macgyver
1	101	2014-02-01	Dallas
2	101	2014-02-02	A-team
6	103	2014-02-01	Matlock

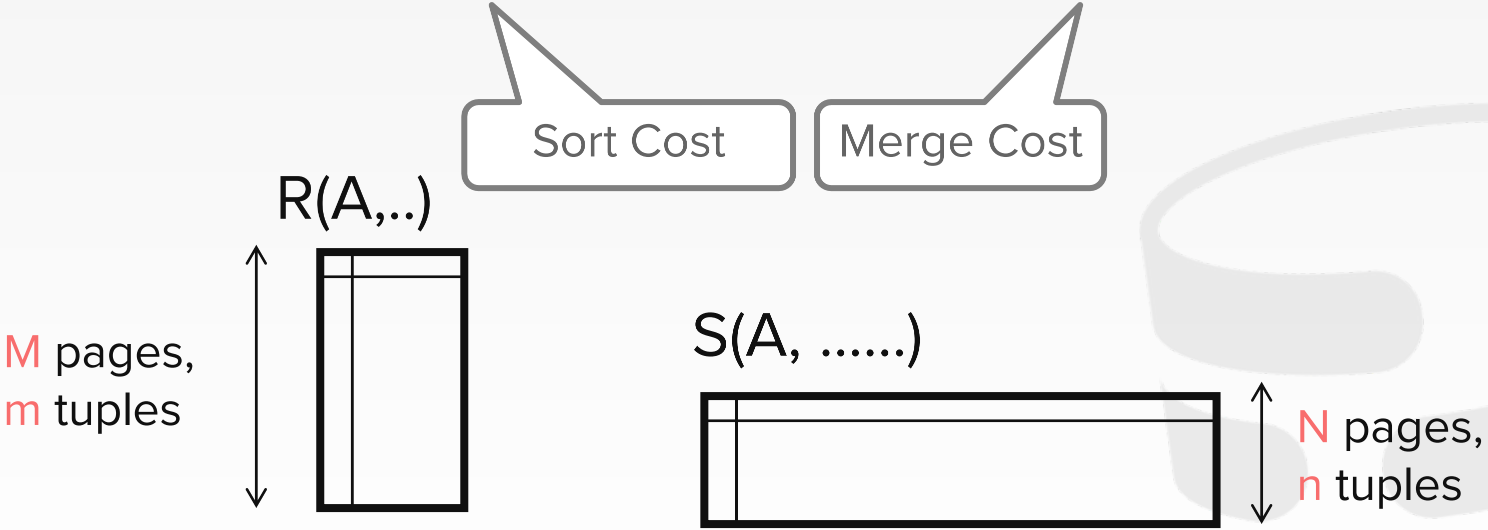
Merge!

Merge!

SORT-MERGE JOIN

Number of disk accesses

→ Cost: [(2M · logM/logB) + (2N · logN/logB)] + [M + N]



SORT-MERGE JOIN

With 100 buffer pages, both R and S can be sorted in 2 passes:

→ Cost: 7,500 I/Os

→ At 0.1 ms/IO, Total time ≈ 0.75 seconds

SORT-MERGE JOIN

Worst case for merging phase?

→ When the join attribute of all of the tuples in both relations contain the same value.

→ Cost: $(M \cdot N)$ + (sort cost)

Andy: Don't worry kids! This is unlikely!

JOIN ALGORITHMS: SUMMARY

JOIN ALGORITHM	I/O COST	TOTAL TIME
Simple Nested Loop Join	$M + (m \cdot N)$	1.3 hours
Block Nested Loop Join	$M + (M \cdot N)$	50 seconds
Index Nested Loop Join	$M + (m \cdot \log N)$	20 seconds
Sort Merge Join	M + N + (sort cost)	0.75 seconds

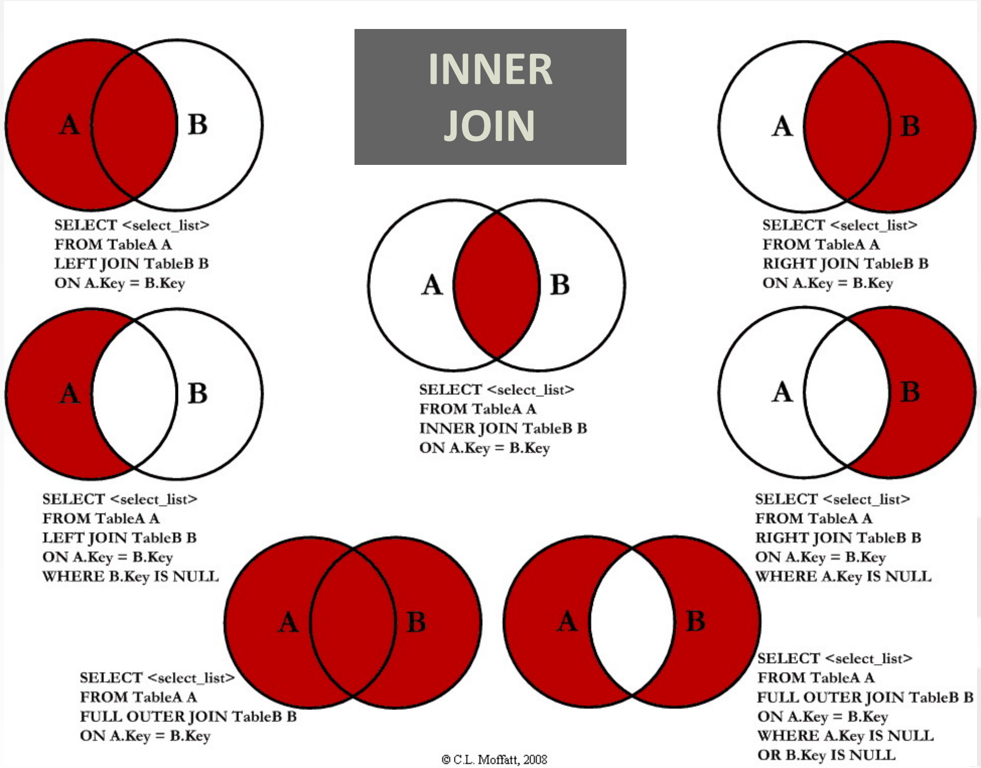
JOIN TYPES

LEFT OUTER JOIN

LEFT OUTER JOIN WITH EXCLUSION

FULL OUTER JOIN

INNER JOIN



RIGHT OUTER JOIN

RIGHT OUTER JOIN WITH EXCLUSION

FULL OUTER JOIN WITH EXCLUSION

CARNEGIE MELLON DATABASE GROUP

CASE STUDY: POSTGRESQL

Employs a state machine to track the **join algorithm's state**

→ At each state, does something and then proceeds to another state

→ State transitions depend on **join type**

PostgreSQL

```
Join {
    get initial outer and inner tuples
    do forever {
        while (outer != inner) {
            if (outer < inner)
                advance outer
            else
                advance inner
        }
        mark inner position
        do forever {
            while (outer == inner) {
                join tuples
                advance inner position
            }
            advance outer position
            if (outer == mark)
                restore inner position to mark
            else
                break    // return to top of outer loop
        }
    }
}
```

```
INITIALIZE
SKIP_TEST
SKIPOUTER_ADVANCE
SKIPINNER_ADVANCE
SKIP_TEST
JOINTUPLES
NEXTINNER
NEXTOUTER
TESTOUTER
TESTOUTER
```

JOIN ALGORITHM STATES

CONCLUSION

There are many join algorithms.

→ Illustrates the sophistication of the technology underlying database systems.

Picking a join algorithm is challenging.

→ Index Nested Loop when selectivity is small.

→ Sort-Merge when joining whole tables.

Stay tuned for more details in next week's query optimization lecture.

RECAP

Sorting algorithms
Join algorithms

NEXT CLASS

Join Algorithms: Hash Join

More Exotic Join Types: Semi, Anti, Lateral

Aggregation Algorithms