Join Algorithms

CREATING THE NEXT®

Administrivia

- Guest lecture on Wednesday
- Extra credit exam on Nov 22 (next Monday)

Today's Agenda

Recap

External Merge Sort

- Divide-and-conquer sorting algorithm that splits the data set into separate **runs** and then sorts them individually.
- **Phase 1 – Sorting**
  - Sort blocks of data that fit in main-memory and then write back the sorted blocks to a file on disk.
- **Phase 2 – Merging**
  - Combine sorted sub-files into a single larger file.

Aggregation

- Collapse multiple tuples into a single scalar value.
- Two implementation choices:
  - Sorting
  - Hashing

Hashing Aggregate

- Populate an **ephemeral hash table** as the DBMS scans the table.
- For each record, check whether there is already an entry in the hash table:
  - `GROUP BY`: Perform aggregate computation.
  - `DISTINCT`: Discard duplicates.
- If everything fits in memory, then it is easy.
- If the DBMS must spill data to disk, then we need to be smarter.

Today's Agenda

- Overview
- Nested Loop Join
- Sort-Merge Join
- Hash Join

Overview

Why do we need to join?

- We **normalize** tables in a relational database to avoid unnecessary repetition of information.
- We use the join operator to reconstruct the original tuples without any information loss.

Denormalized Tables

Artists (<u>ID</u>, Artist, Year, City)
Albums (<u>ID</u>, Album, Artist, Year)

Artists

<u>ID</u>	Artist	Year	City
1	Mozart	1756	Salzburg
2	Beethoven	1770	Bonn

Albums

<u>ID</u>	Album	Artist	Year
1	The Marriage of Figaro	Mozart	1786
2	Requiem Mass In D minor	Mozart	1791
3	Für Elise	Beethoven	1867

Normalized Tables

Artists (ID, Artist, Year, City)
Albums (ID, Album, Year)
ArtistAlbum (Artist_ID, Album_ID)

ArtistAlbum

Artist_ID	Album_ID
1	1
2	1
2	2

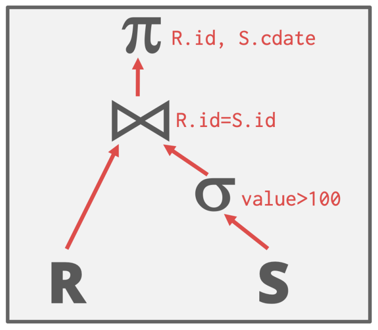
Join Algorithms

- We will focus on combining **two tables** at a time with **inner equi-join** algorithms.
  - These algorithms can be tweaked to support other types of joins.
- In general, we want the smaller table to always be the left table (**outer table**) in the query plan.

Join Operators

- **Decision 1: Output**
  - What data does the join operator emit to its parent operator in the query plan tree?
- **Decision 2: Cost Analysis Criteria**
  - How do we determine whether one join algorithm is better than another?

```
SELECT R.id, S.cdate
  FROM R JOIN S
    ON R.id = S.id
  WHERE S.value > 100
```



Join Operator Output

- For a tuple $r \in R$ and a tuple $s \in S$ that match on join attributes, concatenate r and s together into a new tuple.
- Contents can vary:
  - Depends on query processing model
  - Depends on storage model
  - Depends on the query

Join Operator Output: Data

- Copy the values for the attributes in outer and inner tuples into a new output tuple.
- Subsequent operators in the query plan never need to go back to the base tables to get more data.

R(id,name)

id	name
123	abc

$\bowtie$

S(id,value,cdate)

id	value	cdate
123	1000	10/26/20
123	2000	10/26/20

R.id	R.name	S.id	S.value	S.cdate
123	abc	123	1000	10/26/20
123	abc	123	2000	10/26/20

Join Operator Output: Record Ids

- Only copy the joins keys along with the record ids of the matching tuples.
- Ideal for **column stores** because the DBMS does not copy data that is not need for the query.
- This is called **late materialization**.

R(id,name)

id	name
123	abc

⋈

S(id,value,cdate)

id	value	cdate
123	1000	10/26/20
123	2000	10/26/20

R.id	R.RID	S.id	S.RID
123	R.###	123	S.###
123	R.###	123	S.###

I/O Cost Analysis

- Assume:
  - **M** pages in table **R**, m tuples in R
  - **N** pages in table **S**, n tuples in S
- Cost Metric: Number of IO operations to compute join
- We will ignore output costs (since that depends on the data and we cannot compute that yet).

```
SELECT R.id, S.cdate
  FROM R JOIN S
    ON R.id = S.id
  WHERE S.value > 100
```

Join vs Cross-Product

- $R \bowtie S$ is the most common operation and thus must be carefully optimized.
- $R \times S$ followed by a selection is inefficient because the cross-product is large.
- There are many algorithms for reducing join cost, but no algorithm works well in all scenarios.

Join Algorithms

- Nested Loop Join
  - Naïve
  - Block
  - Index
- Sort-Merge Join
- Hash Join

Nested Loop Join

Nested Loop Join

R (<u>id</u>, name)
S (<u>id</u>, value, cdate)

```
operator NestedLoopJoin(R, S):
    for each tuple r ∈ R: // Outer Table
        for each tuple s ∈ S: // Inner Table
            emit, if r and s match
```

Naïve Nested Loop Join

- Why is this algorithm naïve?
  - For every tuple in **R**, it scans **S** once
- R: M pages, m tuples
- S: N pages, n tuples
- **Cost:** M + (m x N)

Naïve Nested Loop Join

- Example Database:
  - Table R: M = 1000 pages, m = 100,000 tuples
  - Table S: N = 500 pages, n = 40,000 tuples
  - Each page = 4 KB $\Longrightarrow$ Database size = 6 MB
- **Cost Analysis:**
  - M + (m x N) = 1000 + (100000 x 500) = 50,001,000 IOs
  - At 0.1 ms/IO, Total time $\approx$ 1.3 hours
- What if smaller table (S) is used as the outer table?
  - N + (n x M) = 500 + (40000 x 1000) = 40,000,500 IOs
  - At 0.1 ms/IO, Total time $\approx$ 1.1 hours

Block Nested Loop Join

R (id, name)
S (id, value, cdate)

operator BlockNestedLoopJoin(*R, S):*
    for each block $b_R \in$ R: // Outer Table
        for each block $b_S \in$ S: // Inner Table
            for each tuple r $\in b_R$:
                for each tuple s $\in b_S$:
                    emit, if r and s match

Block Nested Loop Join

- This algorithm performs fewer disk accesses.
  - For every block in R, it scans S once
- **Cost:** M + (M x N)

Block Nested Loop Join

- Which one should be the outer table?
  - The smaller table in terms of number of pages

Block Nested Loop Join

- Example Database:
  - Table R: M = 1000 pages, m = 100,000 tuples
  - Table S: N = 500 pages, n = 40,000 tuples
- **Cost Analysis:**
  - M + (M x N) = 1000 + (1000 x 500) = 501,000 IOs
  - At 0.1 ms/IO, Total time ≈ 50 seconds

External Block Nested Loop Join

- What if we have **B** buffers available?
  - Use **B-2** buffers for scanning the outer table.
  - Use one buffer for the inner table, one buffer for storing output.

External Block Nested Loop Join

R (<u>id</u>, name)
S (<u>id</u>, value, cdate)

operator ExternalBlockNestedLoopJoin(*R, S*):
  for each B-2 block $b_R \in$ R: // Outer Table
    for each block $b_S \in$ S: // Inner Table
      for each tuple r $\in b_R$:
        for each tuple s $\in b_S$:
          emit, if r and s match

Block Nested Loop Join

- This algorithm uses B-2 buffers for scanning R.
- **<u>Cost:</u>** $M + (\lceil M / (B-2) \rceil \times N)$
- What if the outer relation completely fits in memory (*i.e.*, B-2 > M)?
  - **<u>Cost:</u>** M + N = 1000 + 500 = 1500 IOs
  - At 0.1 ms/IO, Total time ≈ 0.15 seconds

Nested Loop Join

- Why do basic nested loop joins suck?
  - For each tuple in the outer table, we must do a **sequential scan** to check for a match in the inner table.
- We can avoid sequential scans by using an **index** to find inner table matches.
  - Use an existing index for the join.
  - Or build an index on the fly (*e.g.*, hash table, B+Tree).

Index Nested Loop Join

R (id, name)
S (id, value, cdate)
Index on S (id)

operator IndexNestedLoopJoin(*R, S*):
  for each tuple r ∈ R: // Outer Table
    for each tuple s ∈ Index($r_i = s_i$): // Index on Inner Table
      emit, if r and s match

Index Nested Loop Join

- Assume the cost of each **index probe** is some constant C per tuple.
- **Cost:** M + (m x C)

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 if available.

Sort-Merge Join

Sort-Merge Join

- **Phase 1: Sort**
  - Sort both tables on the join key(s).
- **Phase 2: Merge**
  - We can then use the external merge sort algorithm to join the sorted tables.
  - Step through the two sorted tables with cursors and emit matching tuples.
  - May need to backtrack depending on the join type.

Sort-Merge Join

R (id, name)
S (id, value, cdate)

operator SortMergeJoin(R, S):
  sort R,S on join keys
  $cursor_R \leftarrow R_{sorted}$, $cursorS \leftarrow S_{sorted}$
  while $cursor_R$ and $cursorS$:
    if $cursor_R > cursorS$:
      increment $cursorS$
    else if $cursor_R < cursorS$:
      increment $cursorR$
    else if $cursor_R$ and $cursorS$ match:
      emit
      increment $cursorS$

Sort-Merge Join

R(id,name)

id	name
600	Mark
200	Rahul
100	Maria
300	Li
500	Shiyi
700	Alex
200	Peter
400	Ranveer

S(id,value,cdate)

id	value	cdate
100	2222	10/27/20
500	7777	10/27/20
400	6666	10/27/20
100	9999	10/27/20
200	8888	10/27/20

```
SELECT R.id, S.cdate
  FROM R JOIN S
    ON R.id = S.id
 WHERE S.value > 100
```

Sort-Merge Join

R(id,name)

id	name
600	Mark
200	Rahul
100	Maria
300	Li
500	Shiyi
700	Alex
200	Peter
400	Ranveer

Sort!

S(id,value,cdate)

id	value	cdate
100	2222	10/27/20
500	7777	10/27/20
400	6666	10/27/20
100	9999	10/27/20
200	8888	10/27/20

Sort!

```
SELECT R.id, S.cdate
  FROM R JOIN S
    ON R.id = S.id
 WHERE S.value > 100
```

Sort-Merge Join

R(id,name)

id	name
100	Maria
200	Rahul
200	Peter
300	Li
400	Ranveer
500	Shiyi
600	Mark
700	Alex

Sort!

S(id,value,cdate)

id	value	cdate
100	2222	10/27/20
100	9999	10/27/20
200	8888	10/27/20
400	6666	10/27/20
500	7777	10/27/20

Sort!

```
SELECT R.id, S.cdate
  FROM R JOIN S
    ON R.id = S.id
 WHERE S.value > 100
```

Sort-Merge Join

R(id,name)

id	name
100	Maria
200	Rahul
200	Peter
300	Li
400	Ranveer
500	Shiyi
600	Mark
700	Alex

S(id,value,cdate)

id	value	cdate
100	2222	10/27/20
100	9999	10/27/20
200	8888	10/27/20
400	6666	10/27/20
500	7777	10/27/20

```
SELECT R.id, S.cdate
  FROM R JOIN S
    ON R.id = S.id
 WHERE S.value > 100
```

Output Buffer

R.id	R.name	S.id	S.value	S.cdate
100	Maria	100	2222	10/27/20

Sort-Merge Join

R(id,name)

id	name
100	Maria
200	Rahul
200	Peter
300	Li
400	Ranveer
500	Shiyi
600	Mark
700	Alex

S(id,value,cdate)

id	value	cdate
100	2222	10/27/20
100	9999	10/27/20
200	8888	10/27/20
400	6666	10/27/20
500	7777	10/27/20

```
SELECT R.id, S.cdate
  FROM R JOIN S
    ON R.id = S.id
 WHERE S.value > 100
```

Output Buffer

R.id	R.name	S.id	S.value	S.cdate
100	Maria	100	2222	10/27/20
100	Maria	100	9999	10/27/20

Sort-Merge Join

R(id,name)

id	name
100	Maria
200	Rahul
200	Peter
300	Li
400	Ranveer
500	Shiyi
600	Mark
700	Alex

S(id,value,cdate)

id	value	cdate
100	2222	10/27/20
100	9999	10/27/20
200	8888	10/27/20
400	6666	10/27/20
500	7777	10/27/20

```
SELECT R.id, S.cdate
  FROM R JOIN S
    ON R.id = S.id
 WHERE S.value > 100
```

Output Buffer

R.id	R.name	S.id	S.value	S.cdate
100	Maria	100	2222	10/27/20
100	Maria	100	9999	10/27/20

Sort-Merge Join

R(id,name)

id	name
100	Maria
200	Rahul
200	Peter
300	Li
400	Ranveer
500	Shiyi
600	Mark
700	Alex

S(id,value,cdate)

id	value	cdate
100	2222	10/27/20
100	9999	10/27/20
200	8888	10/27/20
400	6666	10/27/20
500	7777	10/27/20

```
SELECT R.id, S.cdate
  FROM R JOIN S
    ON R.id = S.id
 WHERE S.value > 100
```

Output Buffer

R.id	R.name	S.id	S.value	S.cdate
100	Maria	100	2222	10/27/20
100	Maria	100	9999	10/27/20

Sort-Merge Join

R(id,name)

id	name
100	Maria
200	Rahul
200	Peter
300	Li
400	Ranveer
500	Shiyi
600	Mark
700	Alex

S(id,value,cdate)

id	value	cdate
100	2222	10/27/20
100	9999	10/27/20
200	8888	10/27/20
400	6666	10/27/20
500	7777	10/27/20

```
SELECT R.id, S.cdate
  FROM R JOIN S
    ON R.id = S.id
 WHERE S.value > 100
```

Output Buffer

R.id	R.name	S.id	S.value	S.cdate
100	Maria	100	2222	10/27/20
100	Maria	100	9999	10/27/20
200	Peter	200	8888	10/27/20
200	Peter	200	8888	10/27/20
400	Ranveer	200	6666	10/27/20
500	Shiyi	500	7777	10/27/20

Sort-Merge Join

- Sort Cost (**R**): 2M x (1 + $\lceil log_{B-1} \lceil M / B \rceil \rceil$)
- Sort Cost (**S**): 2N x (1 + $\lceil log_{B-1} \lceil N / B \rceil \rceil$)
- Merge Cost: (M + N)
- **Total Cost:** Sort + Merge

Sort-Merge Join

- Example Database:
  - Table R: M = 1000 pages, m = 100,000 tuples
  - Table S: N = 500 pages, n = 40,000 tuples
- With B=100 buffer pages, both R and S can be sorted in two passes:
  - Sort Cost (R) = 2000 x (1 + $\lceil log_{99}$ 1000 /100$\rceil$) = 4000 IOs
  - Sort Cost (S) = 1000 x (1 + $\lceil log_{99}$ 500 / 100$\rceil$) = 2000 IOs
  - Merge Cost = (1000 + 500) = 1500 IOs
  - Total Cost = 4000 + 2000 + 1500 = 7500 IOs
  - At 0.1 ms/IO, Total time ≈ 0.75 seconds

Sort-Merge Join

- The worst case for the merging phase is when the join attribute of all of the tuples in both relations contain the **same value**.
- **Cost:** (M x N) + (sort cost)

When is Sort-Merge Join Useful?

- One or both tables are already sorted on join key.
- Output must be sorted on join key.
- The input relations may be sorted by either by an explicit sort operator, or by scanning the relation using an index on the join key.

Hash Join

Hash Join

- If tuple r ∈ R and a tuple s ∈ S satisfy the join condition, then they have the same value for the join attributes.
- If that value is hashed to some partition i, the R tuple must be in r_i and the S tuple in s_i.
- Therefore, R tuples in r_i need only to be compared with S tuples in s_i.

Basic Hash Join Algorithm

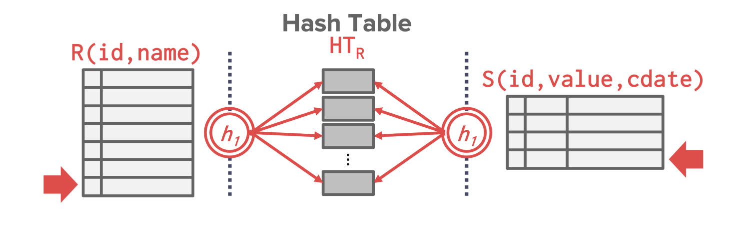
- **Phase 1: Build**
  - Scan the outer table and populate a hash table using the hash function h_1 on the join attributes.
- **Phase 2: Probe**
  - Scan the inner table and use h_1 on each tuple to jump to a location in the hash table and find a matching tuple.

Basic Hash Join Algorithm

R (<u>id</u>, name)
S (<u>id</u>, value, cdate)

operator BasicHashJoin(*R, S*):
  build hash table HT_R for R
  for each tuple s $\in$ S
    emit, if $h_1(s)$ in HT_R

Basic Hash Join Algorithm



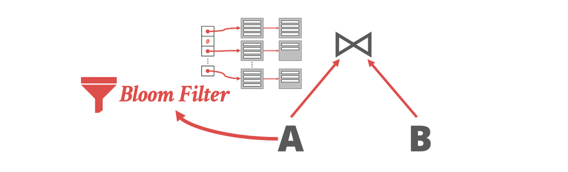
Hash Table Contents

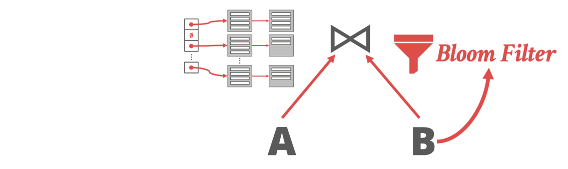
- **Key:** The attribute(s) that the query is joining the tables on.
- **Value:** Depends on what the parent operator above the join in the query plan expects as its input.
  - **Approach 1: Full Tuple**
    - Avoid having to retrieve the outer table's tuple contents on a match.
    - Takes up more space in memory.
  - **Approach 2: Tuple Identifier**
    - Ideal for column stores because the DBMS does **not** fetch data from disk unless needed.
    - Also better if join selectivity is low.

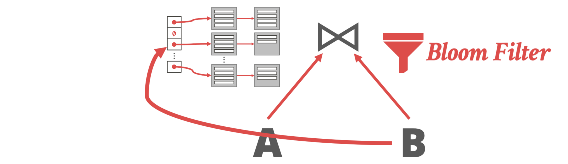
Probe Phase Optimization

- Create a **bloom filter** during the build phase when the key is likely to **not** exist in the hash table.
  - Threads check the filter before probing the hash table.
  - This will be faster since the filter will fit in CPU caches.
  - *a.k.a.*, sideways information passing.

Probe Phase Optimization





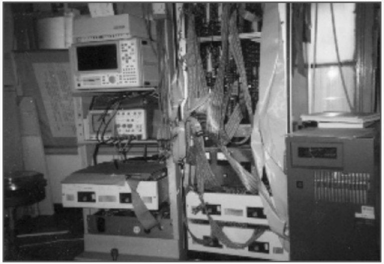


Hash Join

- What happens if we do not have enough memory to fit the entire hash table?
- We do **<u>not</u>** want to let the buffer pool manager swap out the hash table pages randomly.

Grace Hash Join

- Hash join when tables do **not** fit in memory.
  - **Build Phase:** Hash both tables on the join attribute into partitions.
  - **Probe Phase:** Compares tuples in corresponding partitions for each table.
- Named after the **GRACE database machine** from Japan in the 1980s.



GRACE
University of Tokyo

Grace Hash Join

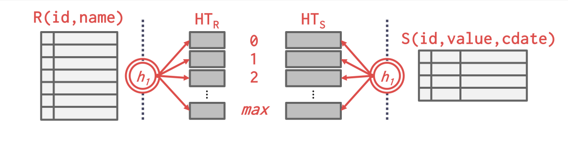
- Hash **R** into (0, 1, ..., *max*) buckets.
- Hash **S** into the same number of buckets with the same hash function.
- Join each pair of matching buckets between R and S.

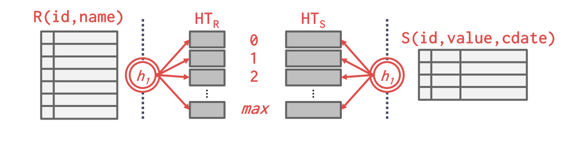
Grace Hash Join

R (<u>id</u>, name)
S (<u>id</u>, value, cdate)

operator Grace Hash Join(*R, S):*
  for bucket i ∈ [0, *max*]
    for each tuple r ∈ bucket R_i
      for each tuple s ∈ bucket S_i
        emit, if r and s match

Grace Hash Join

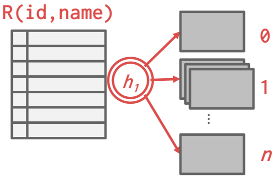




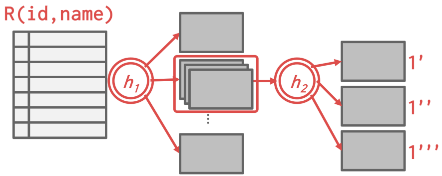
Grace Hash Join

- If the buckets do not fit in memory, then use **recursive partitioning** to split the tables into chunks that will fit.
  - Build another hash table for $bucket_{R,i}$ using hash function h_2 (with $h_2 \mathrel{!}= h_1$).
  - Then probe it for each tuple of the other table's bucket at that level.

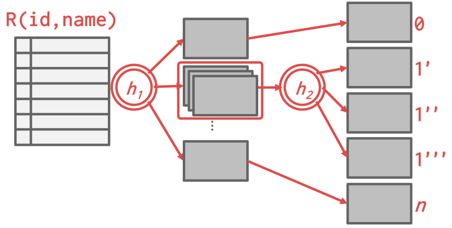
Recursive Partitioning



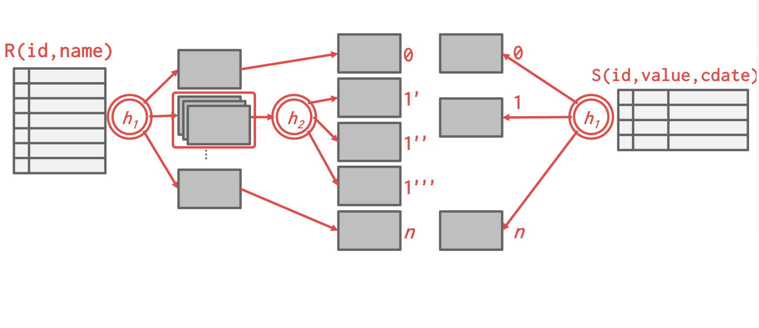
Recursive Partitioning



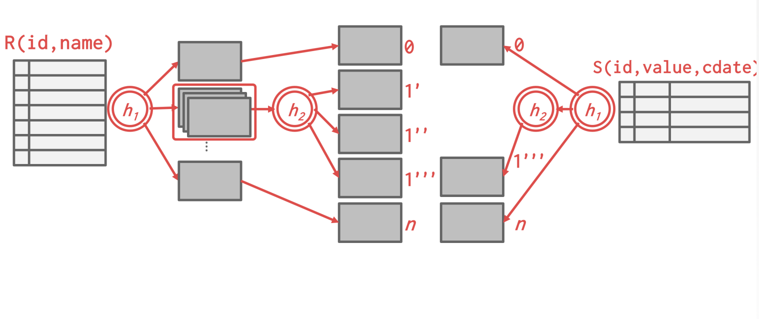
Recursive Partitioning



Recursive Partitioning



Recursive Partitioning



Grace Hash Join

- **Partitioning Phase:**
  - Read+Write both tables
  - 2 x (M + N) IOs
- **Probing Phase:**
  - Read both tables
  - M + N IOs
- **Total Cost:** 3 x (M + N)

Grace Hash Join

- Example Database:
  - Table R: M = 1000 pages, m = 100,000 tuples
  - Table S: N = 500 pages, n = 40,000 tuples
- Cost Analysis:
  - 3 x (M + N) = 3 x(1000 + 500) = 4,500 IOs
  - At 0.1 ms/IO, Total time $\approx$ 0.45 seconds

Observation

- If the DBMS knows the size of the outer table, then it can use a **static hash table**.
  - Less computational overhead for build / probe operations.
- If we do not know the size, then we have to use a **dynamic hash table** or allow for overflow pages.

Conclusion

Join Algorithms: Summary

Join Algorithm	IO Cost	Example
Simple Nested Loop Join	M + (m x N)	1.3 hours
Block Nested Loop Join	M + (M x N)	50 seconds
Index Nested Loop Join	M + (M x C)	Variable
Sort-Merge Join	M + N + (sort cost)	0.75 seconds
Hash Join	3 x (M + N)	0.45 seconds

Conclusion

- Hashing is almost always better than sorting for operator execution.
- Caveats:
  - Sorting is better on non-uniform data.
  - Sorting is better when result needs to be sorted.
- Good DBMSs use either or both.
- Next Class
  - Composing operators together to execute queries.