Carnegie Mellon University

Hash Joins & Aggregations

Lecture #12

Database Systems
15-445/15-645
Fall 2017

Andy Pavlo
Computer Science Dept.
Carnegie Mellon Univ.

ADMINISTRIVIA

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

Mid-term Exam is on Wednesday October 18th (in class)

Project #2 is due Wednesday October 25th @ 11:59am

PostgreSQL 10 Released

Posted on **2017-10-05**

The PostgreSQL Global Development Group today announced the release of PostgreSQL 10, the latest version of the world's most advanced open source database.

A critical feature of modern workloads is the ability to distribute data across many nodes for faster access, management, and analysis, which is also known as a "divide and conquer" strategy. The PostgreSQL 10 release includes significant enhancements to effectively implement the divide and conquer strategy, including native logical replication, declarative table partitioning, and improved query parallelism.

"Our developer community focused on building features that would take advantage of modern infrastructure setups for distributing workloads," said Magnus Hagander, a core team member of the PostgreSQL Global Development Group. "Features such as logical replication and improved query parallelism represent years of work and demonstrate the continued dedication of the community to ensuring Postgres leadership as technology demands evolve."

This release also marks the change of the versioning scheme for PostgreSQL to a "x.y" format. This means the next minor release of PostgreSQL will be 10.1 and the next major release will be 11.

Logical Replication - A publish/subscribe framework for distributing data

Logical replication extends the current replication features of PostgreSQL with the ability to send modifications on a per-database and per-table level to different PostgreSQL databases. Users can now fine-tune the data replicated to various database clusters and will have the ability to perform zero-

LAST CLASS

External Merge Sort

Join Algorithms
→ Nested Loop Join
→ Sort-Merge Join

JOIN ALGORITHMS

There are essentially three classes of join algorithms:
→ Nested Loop
→ Sort-Merge
→ Hash

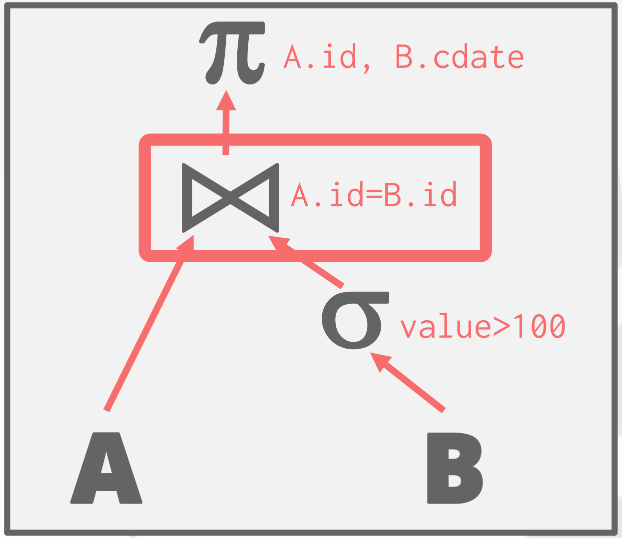
In general, we want the smaller table to always be the outer table.

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 processing model
→ Depends on storage model
→ Depends on the query

```
SELECT A.id, B.cdate
  FROM A, B
 WHERE A.id = B.id
   AND B.value > 100
```



CARNEGIE MELLON DATABASE GROUP

JOIN OPERATOR OUTPUT: DATA

Copy the values for the attributes in outer and inner tuples into a new output tuple.

```
SELECT A.id, B.cdate
  FROM A, B
 WHERE A.id = B.id
   AND B.value > 100
```

JOIN OPERATOR OUTPUT: DATA

Copy the values for the attributes in outer and inner tuples into a new output tuple.

```
SELECT A.id, B.cdate
  FROM A, B
 WHERE A.id = B.id
   AND B.value > 100
```

A(id,name)

id	name
123	abc

B(id,value,cdate)

id	value	cdate
123	1000	10/16/2017
123	2000	10/16/2017

JOIN OPERATOR OUTPUT: DATA

Copy the values for the attributes in outer and inner tuples into a new output tuple.

```
SELECT A.id, B.cdate
  FROM A, B
 WHERE A.id = B.id
   AND B.value > 100
```

A(id,name)

id	name
123	abc

⋈

B(id,value,cdate)

id	value	cdate
123	1000	10/16/2017
123	2000	10/16/2017

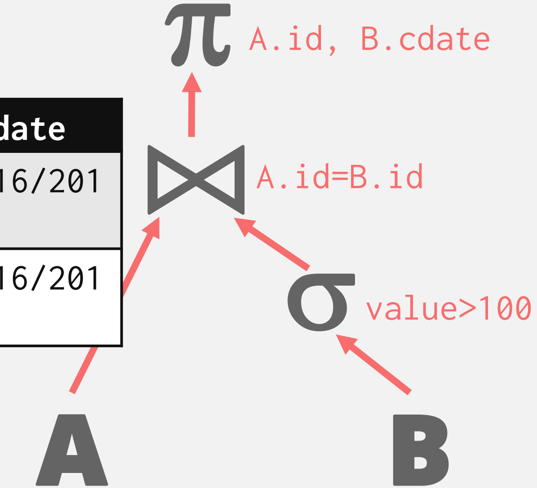
A.id	A.name	B.id	B.value	B.cdate
123	abc	123	1000	10/16/2017
123	abc	123	2000	10/16/2017

JOIN OPERATOR OUTPUT: DATA

Copy the values for the attributes in outer and inner tuples into a new output tuple.

```
SELECT A.id, B.cdate
  FROM A, B
 WHERE A.id = B.id
   AND B.value > 100
```

A.id	A.name	B.id	B.value	B.cdate
123	abc	123	1000	10/16/2017
123	abc	123	2000	10/16/2017

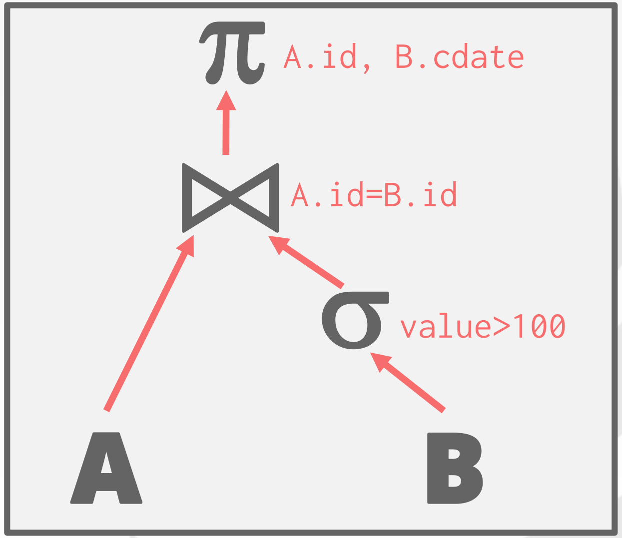


JOIN OPERATOR OUTPUT: DATA

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.

```sql
SELECT A.id, B.cdate
  FROM A, B
 WHERE A.id = B.id
   AND B.value > 100
```



CARNEGIE MELLON DATABASE GROUP

JOIN OPERATOR OUTPUT: RECORD IDS

Only copy the joins keys along with the record ids of the matching tuples.

```
SELECT A.id, B.cdate
  FROM A, B
 WHERE A.id = B.id
   AND B.value > 100
```

A(id,name)

id	name
123	abc

B(id,value,cdate)

id	value	cdate
123	1000	10/16/2017
123	2000	10/16/2017

JOIN OPERATOR OUTPUT: RECORD IDS

Only copy the joins keys along with the record ids of the matching tuples.

```
SELECT A.id, B.cdate
  FROM A, B
 WHERE A.id = B.id
   AND B.value > 100
```

A(id,name)

id	name
123	abc

⋈

B(id,value,cdate)

id	value	cdate
123	1000	10/16/2017
123	2000	10/16/2017

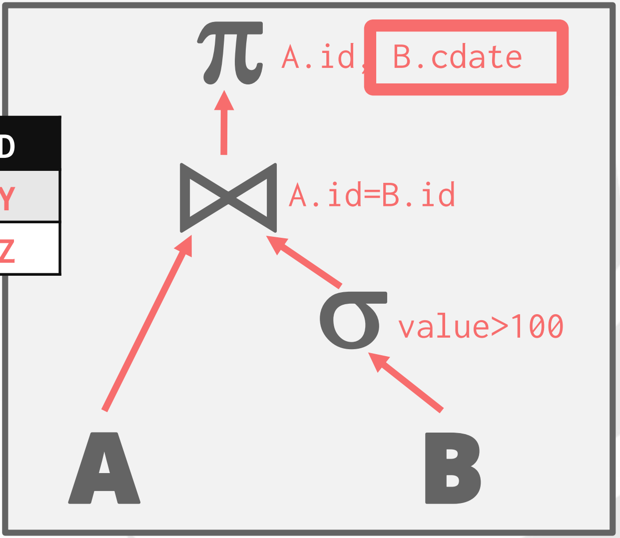
A.id	A.RID	B.id	B.RID
123	A.XXX	123	B.YYY
123	A.XXX	123	B.ZZZ

JOIN OPERATOR OUTPUT: RECORD IDS

Only copy the joins keys along with the record ids of the matching tuples.

```sql
SELECT A.id, B.cdate
  FROM A, B
 WHERE A.id = B.id
   AND B.value > 100
```

A.id	A.RID	B.id	B.RID
123	A.XXX	123	B.YYY
123	A.XXX	123	B.ZZZ



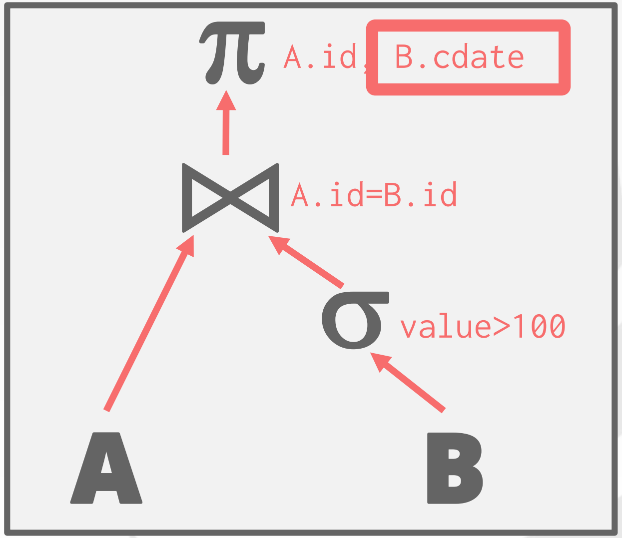
JOIN OPERATOR OUTPUT: RECORD IDS

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**.

```sql
SELECT A.id, B.cdate
  FROM A, B
 WHERE A.id = B.id
   AND B.value > 100
```



TODAY'S AGENDA

Hash Joins

Aggregations

HASH JOIN

If tuple $r \in R$ and a tuple $s \in S$ satisfy the join condition, then they have the same value for the join attributes.

If that value is hashed to some value i, the R tuple has to 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

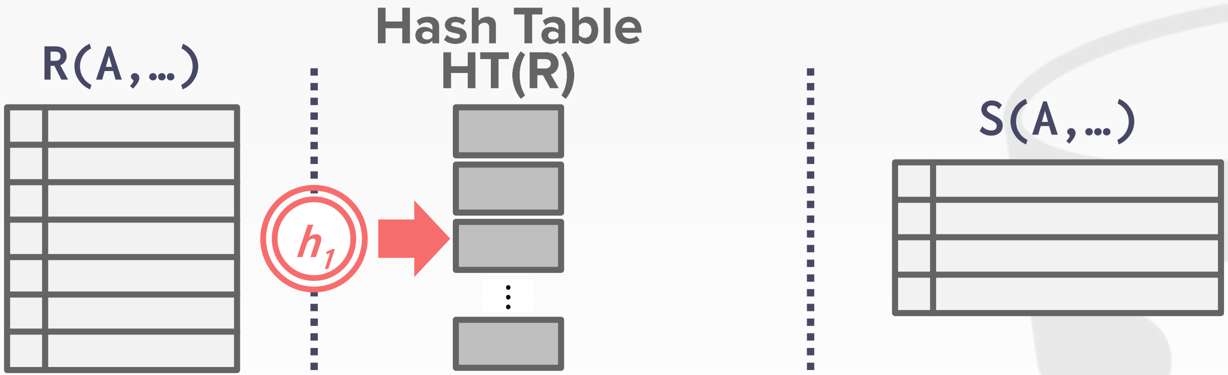
→ Scan the outer relation and populate a hash table using the hash function h_1 on the join attributes.

Phase #2: Probe

→ Scan the inner relation and use h_1 on each tuple to jump to a location in the hash table and find a matching tuple.

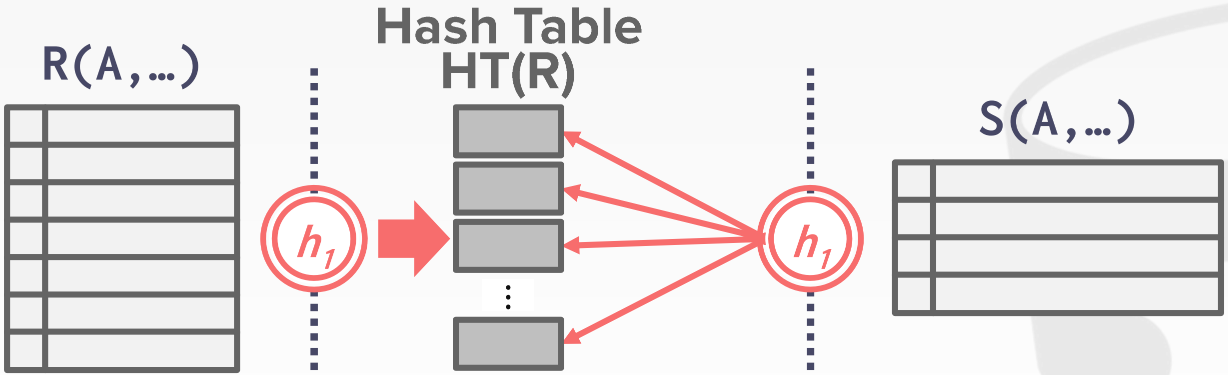
BASIC HASH JOIN ALGORITHM

```
build hash table H for R
foreach tuple s of S
  output, if H1(sj) ∈ HT(R)
```



BASIC HASH JOIN ALGORITHM

```
build hash table H for R
foreach tuple s of S
   output, if H1(sj) ∈ HT(R)
```



CARNEGIE MELLON DATABASE GROUP

HASH TABLE CONTENTS

Key: The attribute(s) that the query is joining the tables on.

Value: Varies per implementation.

→ Depends on what the operators above the join in the query plan expect as its input.

HASH TABLE VALUES

Approach #1: Full Tuple

→ Avoid having to retrieve the outer relation's tuple contents on a match.

→ Takes up more space in memory.

Approach #2: Tuple Identifier

→ Ideal for column stores because the DBMS doesn't fetch data from disk it doesn't need.

→ Also better if join selectivity is low.

HASH JOIN

What happens if we don't have enough memory to fit the entire hash table?

We don't want to let the buffer pool manager swap out the hash table pages at a random.

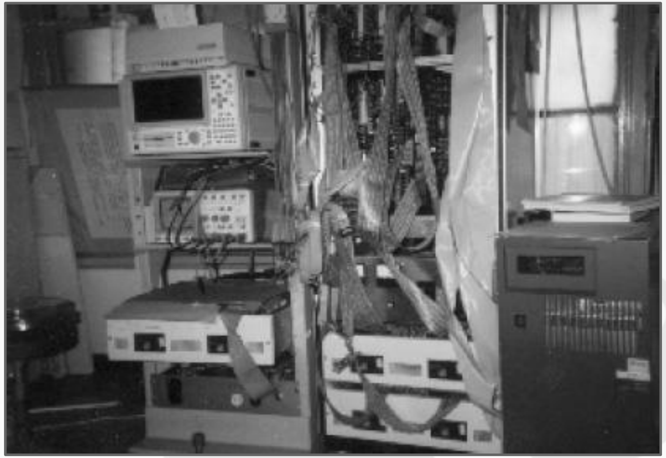
GRACE HASH JOIN

Hash join when tables don't fit in memory.

→ **Build Phase:** Hash both tables on the join attribute into partitions.

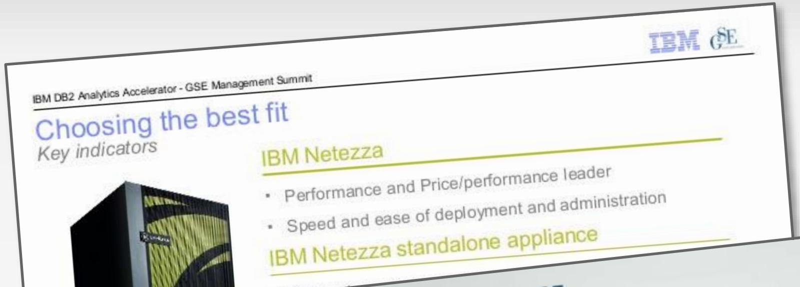
→ **Probe Phase:** Compares tuples in corresponding partitions for each table.

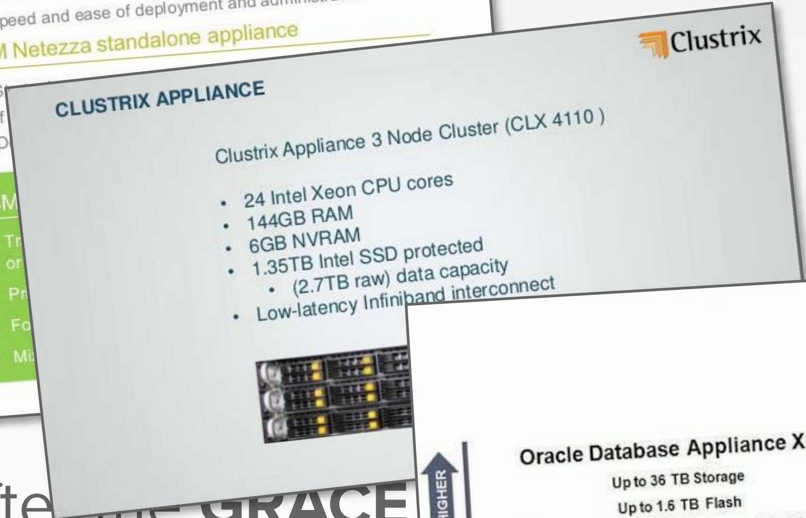
Named after the **GRACE** database machine from Japan.

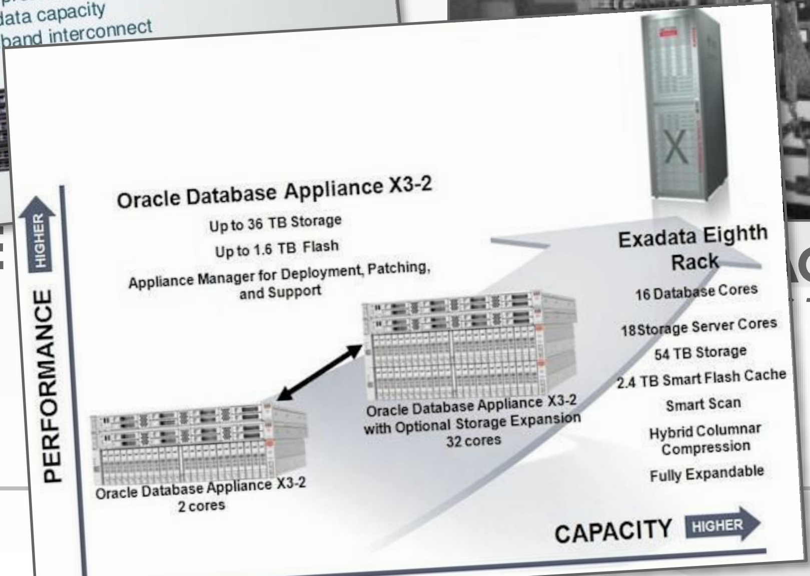


GRACE
Univ. of Tokyo

Named after the **GRACE** machine from Japan.







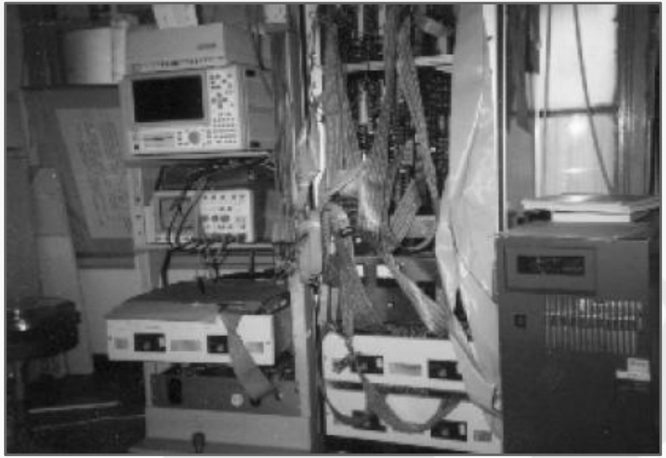
GRACE HASH JOIN

Hash join when tables don't 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.

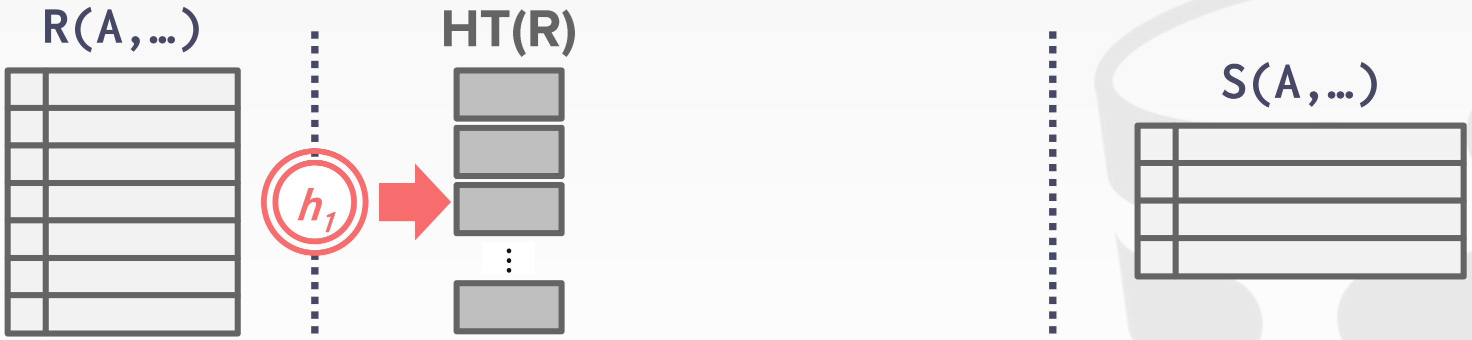


GRACE
Univ. of Tokyo

GRACE HASH JOIN

Hash R into (0, 1, ..., *max*) buckets.

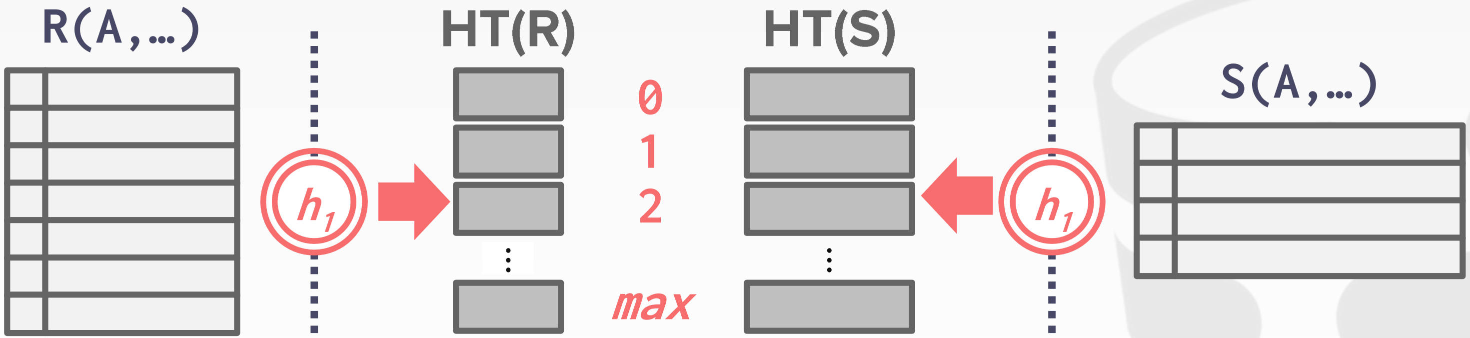
Hash S into the same # of buckets with the same hash function.



GRACE HASH JOIN

Hash R into (0, 1, ..., *max*) buckets.

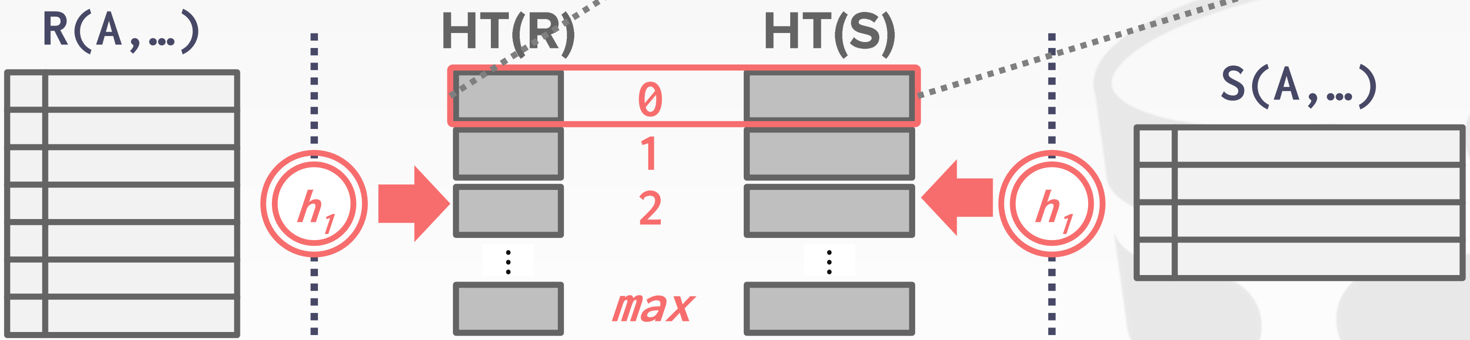
Hash S into the same # of buckets with the same hash function.



GRACE HASH JOIN

Join each pair of matching buckets between **R** and **S**.

```
foreach tuple r ∈ bucket_R,0
  foreach tuple s ∈ bucket_S,0
    output, if match(r, s)
```



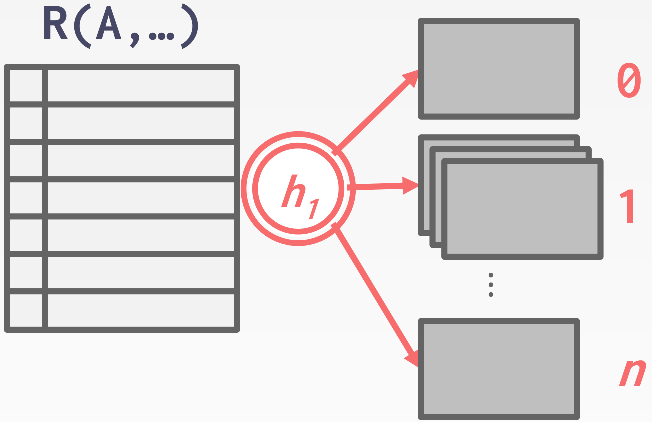
GRACE HASH JOIN

If the buckets don't fit in memory, then use **recursive partitioning**.

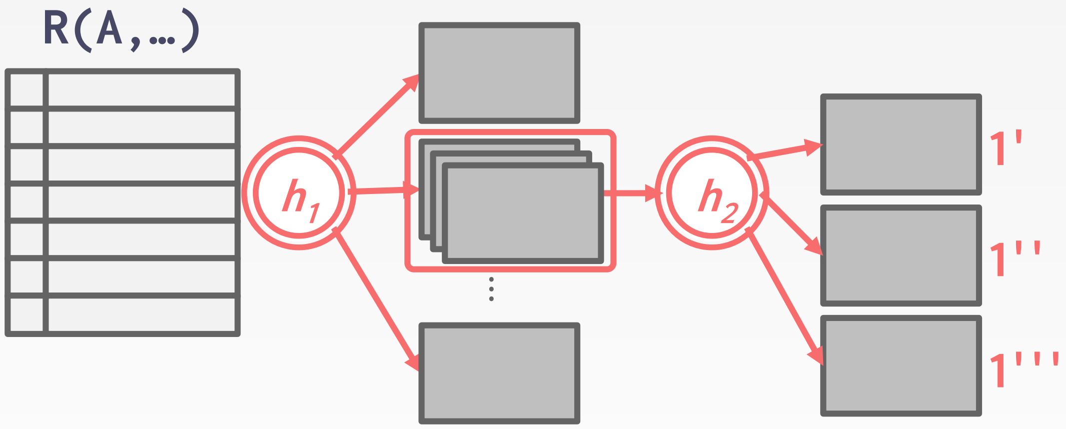
Build another hash table for $\mathbf{bucket}_{R,i}$ using hash function h_2 (with $h_2 \neq h_1$).

Then probe it for each tuple of the other table's bucket at that level.

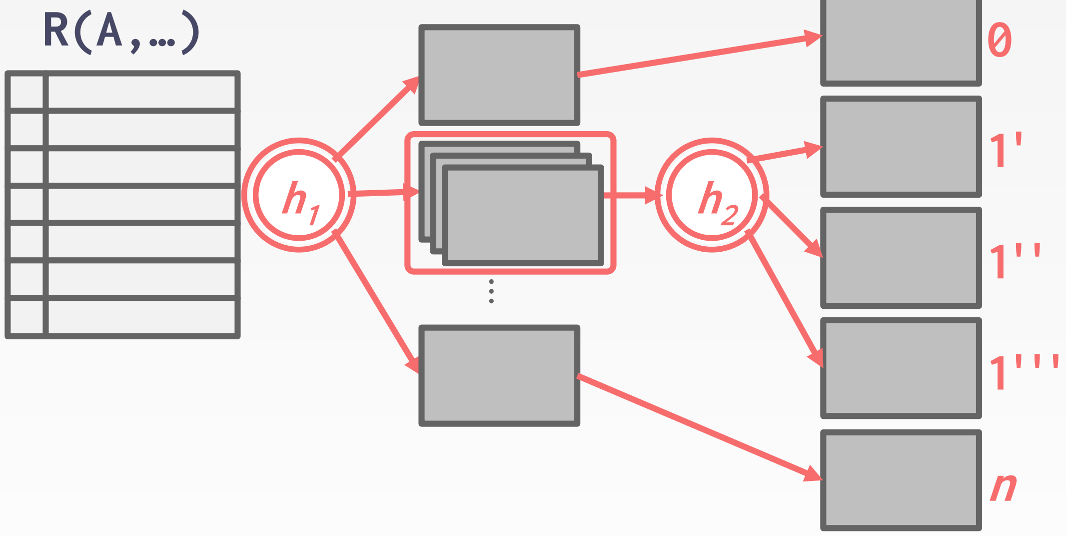
RECURSIVE PARTITIONING



RECURSIVE PARTITIONING

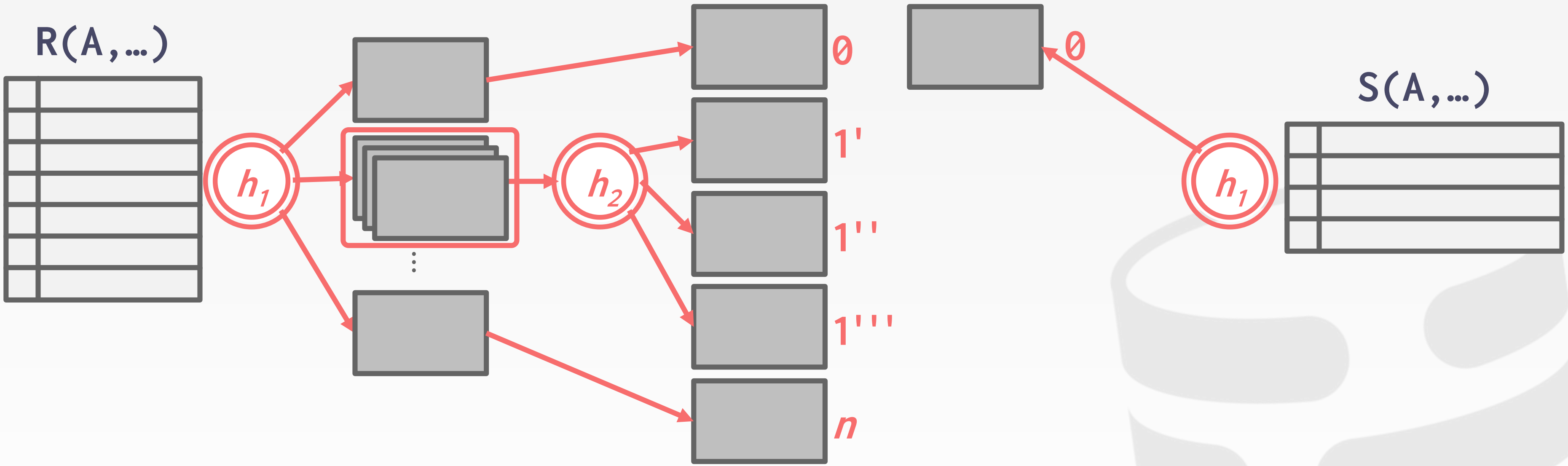


RECURSIVE PARTITIONING

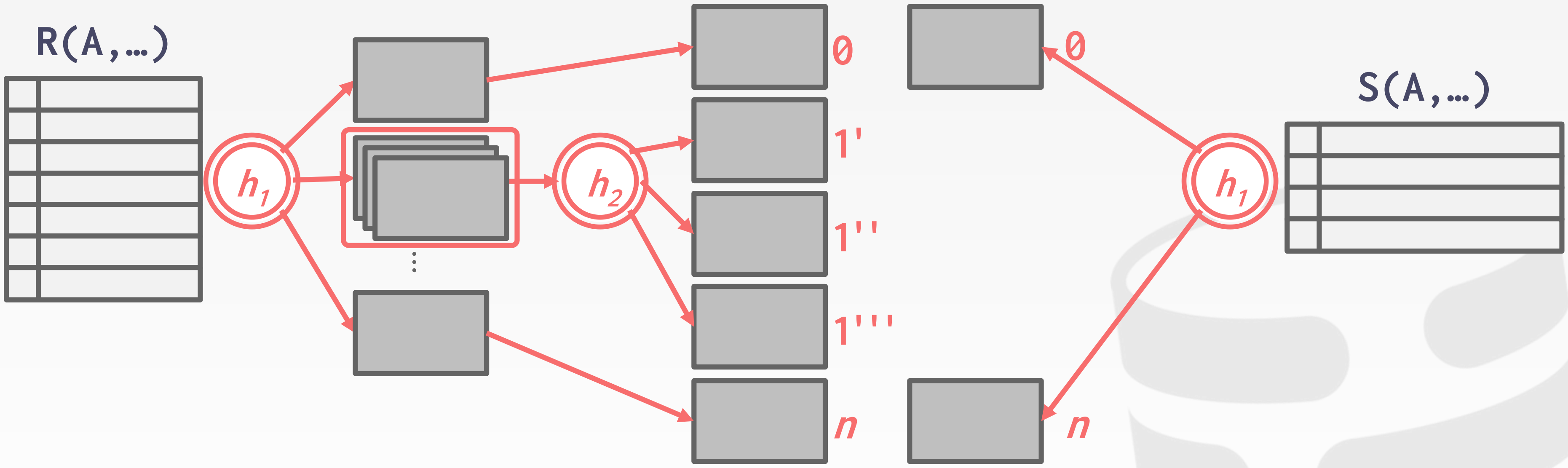


CARNEGIE MELLON DATABASE GROUP

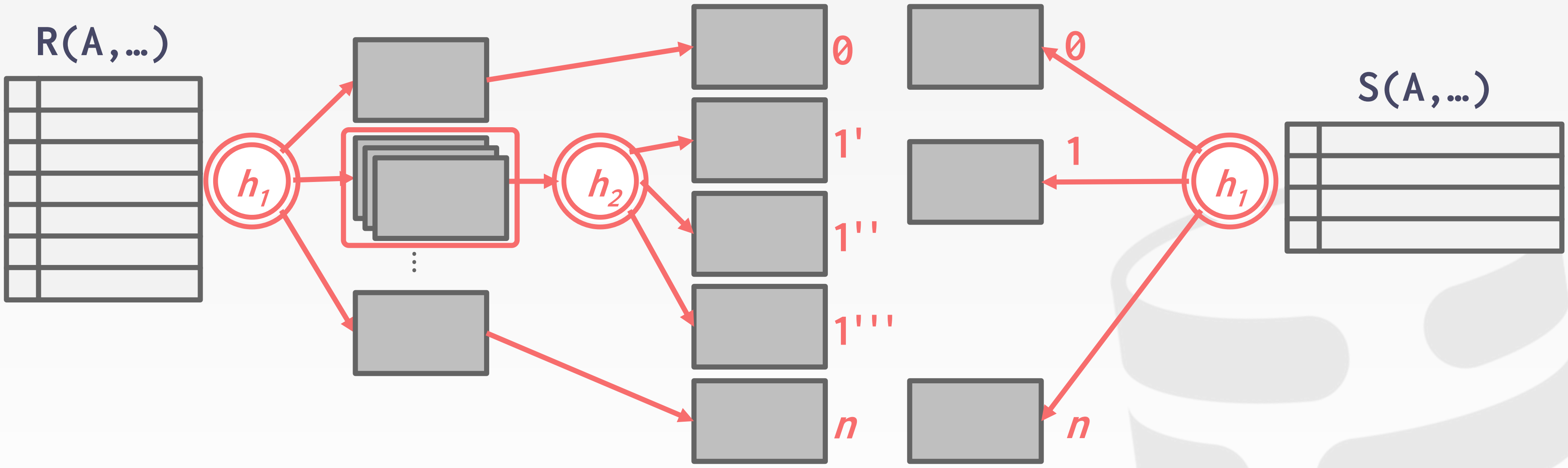
RECURSIVE PARTITIONING



RECURSIVE PARTITIONING

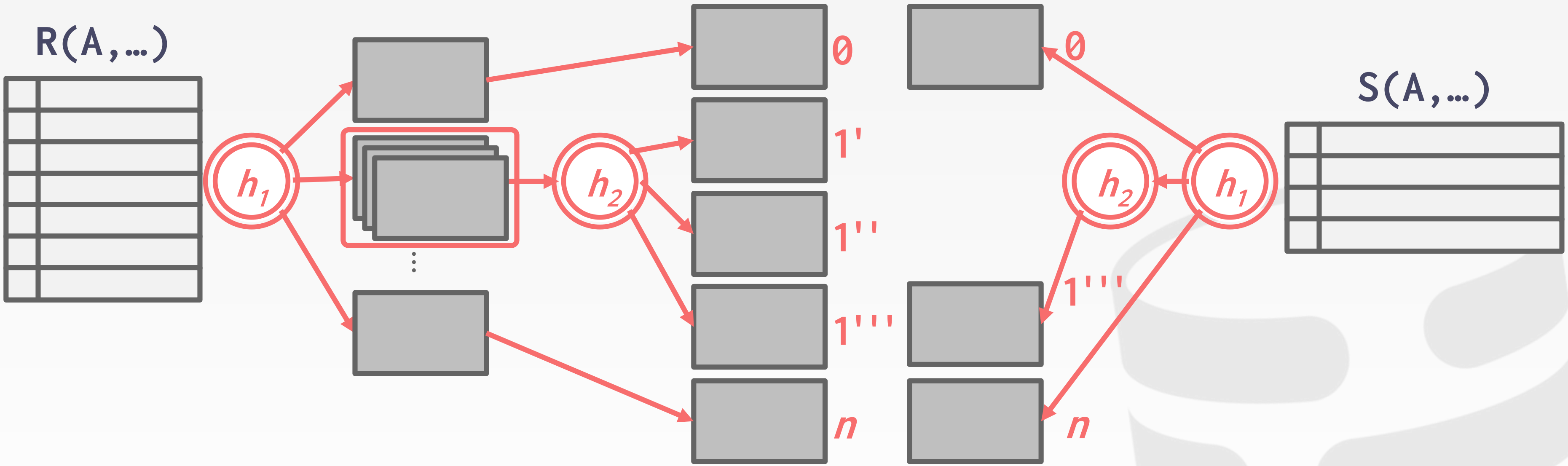


RECURSIVE PARTITIONING



CARNEGIE MELLON DATABASE GROUP

RECURSIVE PARTITIONING



CARNEGIE MELLON DATABASE GROUP

GRACE HASH JOIN

Cost of hash join?

- → Assume that we have enough buffers.
- → Cost: 3(M+N)

Partitioning Phase:

- → Read+Write both tables
- → 2(M+N) I/Os

Probing Phase:

- → Read both tables
- → M+N I/Os

```
M=1000 N=500
3(M+N) = 3•(1000 + 500)
       = 4500 I/Os
```

At 0.1ms/IO ≈ 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 it doesn't know the size, then it has to use a dynamic hash table or allow for overflow pages.

JOIN ALGORITHMS: SUMMARY

JOIN ALGORITHM	I/O COST	TOTAL TIME
Simple Nested Loop Join	M + (M·N)	1.3 hours
Block Nested Loop Join	M + (M·N)	50 seconds
Index Nested Loop Join	M + (M·log N)	20 seconds
Sort-Merge Join	M + N + (sort cost)	0.75 seconds
Hash Join	3(M+N)	0.45 seconds

AGGREGATIONS

Collapse multiple tuples into a single scalar value.

Two implementation choices:
→ Sorting
→ Hashing

SORTING AGGREGATION

```
SELECT DISTINCT cid
  FROM enrolled
 WHERE grade IN ('B','C')
```

enrolled(sid,cid,grade)

sid	cid	grade
53666	15-445	C
53688	15-721	A
53688	15-826	B
53666	15-721	C
53655	15-445	C

Filter

sid	cid	grade
53666	15-445	C
53688	15-826	B
53666	15-721	C
53655	15-445	C

Remove Columns

cid
15-445
15-826
15-721
15-445

Sort

cid
15-445
15-445
15-721
15-826

SORTING AGGREGATION

```
SELECT DISTINCT cid
  FROM enrolled
 WHERE grade IN ('B','C')
```

enrolled(sid,cid,grade)

sid	cid	grade
53666	15-445	C
53688	15-721	A
53688	15-826	B
53666	15-721	C
53655	15-445	C

Filter

sid	cid	grade
53666	15-445	C
53688	15-826	B
53666	15-721	C
53655	15-445	C

Remove Columns

cid
15-445
15-826
15-721
15-445

Sort

cid
15-445
~~15-445~~
15-721
15-826

Eliminate Dupes

SORTING VS. HASHING

What if we don't need the order of the sorted data?

→ Forming groups in **GROUP BY**

→ Removing duplicates in **DISTINCT**

Hashing does this!

→ And may be cheaper than sorting!

→ But what if table doesn't fit in memory?

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:

→ DISTINCT: Discard duplicate.
→ GROUP BY: Perform aggregate computation.

Two phase approach.

HASHING AGGREGATE PHASE #1: PARTITION

Use a hash function **h_1** to split tuples into partitions on disk.

→ We know that all matches live in the same partition.

→ Partitions are "spilled" to disk via output buffers.

Assume that we have **B** buffers.

HASHING AGGREGATE PHASE #1: PARTITION

```
SELECT DISTINCT cid
  FROM enrolled
 WHERE grade IN ('B','C')
```

enrolled(sid,cid,grade)

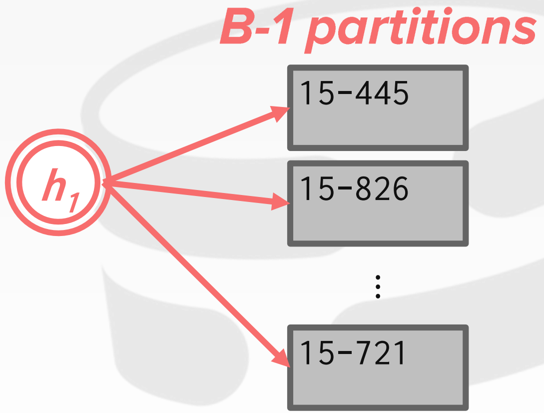
sid	cid	grade
53666	15-445	C
53688	15-721	A
53688	15-826	B
53666	15-721	C
53655	15-445	C

Filter

sid	cid	grade
53666	15-445	C
53688	15-826	B
53666	15-721	C
53655	15-445	C

Remove Columns

cid
15-445
15-826
15-721
15-445



HASHING AGGREGATE PHASE #2: REHASH

For each partition on disk:

→ Read it into memory and build an in-memory hash table based on a second hash function h_2.
→ Then go through each bucket of this hash table to bring together matching tuples.

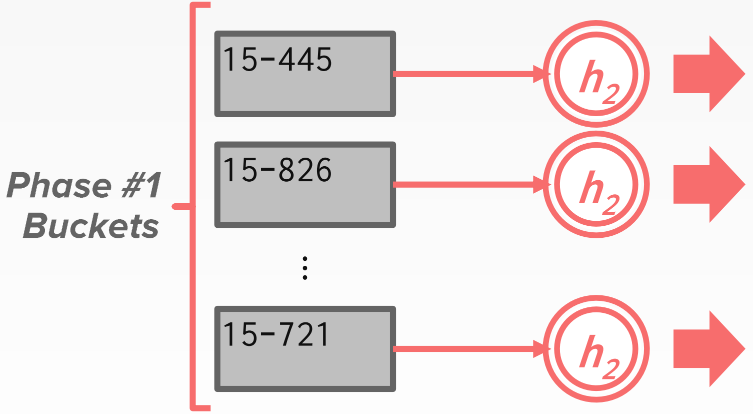
This assumes that each partition fits in memory.

HASHING AGGREGATE PHASE #2: REHASH

```
SELECT DISTINCT cid
  FROM enrolled
 WHERE grade IN ('B','C')
```

enrolled(sid,cid,grade)

sid	cid	grade
53666	15-445	C
53688	15-721	A
53688	15-826	B
53666	15-721	C
53655	15-445	C



Hash Table

Key	Value
XXX	15-445
YYY	15-826
ZZZ	15-721

cid
15-445
15-721
15-826

COST ANALYSIS

How big of a table can we hash using this approach?

→ B-1 "spill partitions" in Phase #1

→ Each should be no more than B blocks big

Answer: $B \cdot (B-1)$

→ A table of N blocks needs about sqrt(N) buffers

→ Assumes hash distributes records evenly!
Use a "fudge factor" $f>1$ for that: we need $B \cdot \text{sqrt}(f \cdot N)$

COST ANALYSIS

If the hash table doesn't fit into memory, then we can use recursive partitioning again.

→ In the ReHash Phase, if a partition `i` is bigger than `B`, then recurse.

→ Pretend that `i` is a table we need to hash, run the Partitioning Phase on `i`, and then the ReHash Phase on each of its (sub)partitions

SORTING VS. HASHING

We can hash a table of size **N** blocks in **sqrt(N)** space.

How big of a table can we sort in 2 passes?

- → Get **N/B** sorted runs after Pass 0
- → Can merge all runs in Pass 1 if **N/B ≤ B-1**
- → Thus, we (roughly) require: **N ≤ B2**
- → We can sort a table of size **N** blocks in about space **sqrt(N)**

SORTING VS. HASHING

Choice of sorting vs. hashing is subtle and depends on optimizations done in each case.

We already discussed the optimizations for sorting:

→ Chunk I/O into large blocks to amortize seek+RD costs.
→ Double-buffering to overlap CPU and I/O.

HASHING SUMMARIZATION

Combine the summarization into the hashing process.

Maintain running totals for each group as you build the hash table.

HASHING SUMMARIZATION

During the ReHash phase, store pairs of the form (`GroupKey`→`RunningVal`)

When we want to insert a new tuple into the hash table:

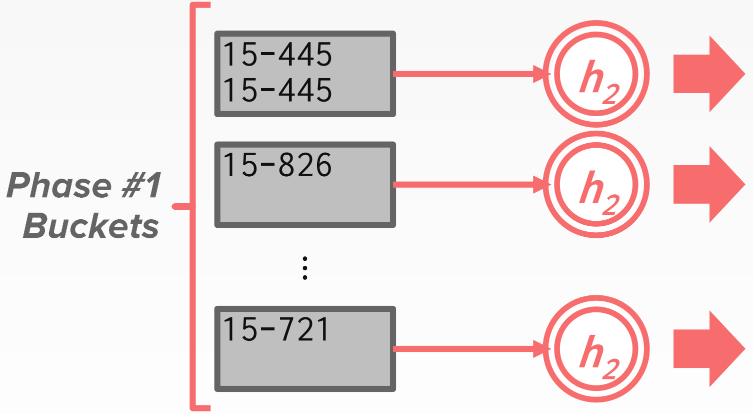
→ If we find a matching `GroupKey`, just update the `RunningVal` appropriately
→ Else insert a new `GroupKey`→`RunningVal`

HASHING SUMMARIZATION

```
SELECT cid, AVG(s.gpa)
  FROM student AS s, enrolled AS e
 WHERE s.sid = e.sid
 GROUP BY cid
```

Running Totals

AVG(col) → (COUNT,SUM)
MIN(col) → (MIN)
MAX(col) → (MAX)
SUM(col) → (SUM)
COUNT(col) → (COUNT)



Hash Table

key	value
XXX	15-445→(2,7.32)
YYY	15-826→(1,3.33)
ZZZ	15-721→(1,2.89)

Final Result

cid	AVG(gpa)
15-445	3.66
15-826	3.33
15-721	2.89

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

How the DBMS decides what algorithm to use for each operator in a query plan.