Relational Algebra, Iterators, & Operator Implementations

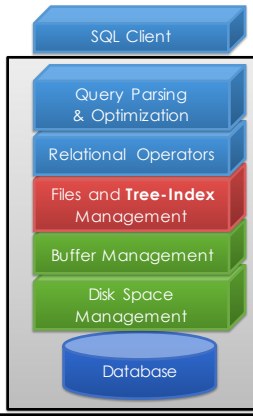
R & G, Chapters
- Relational Algebra: 4.1 - 4.2
- Relational → Query Plan: 12
- Select, Project, **Join** operators: 14

Berkeley cs186

Architecture of a DBMS

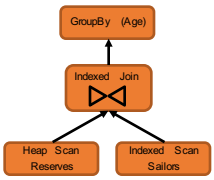
Last lecture:
- Finished **tree indexes**: ISAM and *B+-Trees*

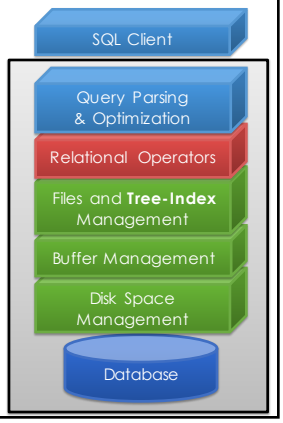
Today:
Relational Operators



Architecture of a DBMS

Execute the dataflow by operating on **records** and **files**

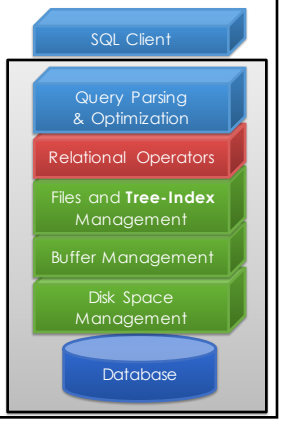




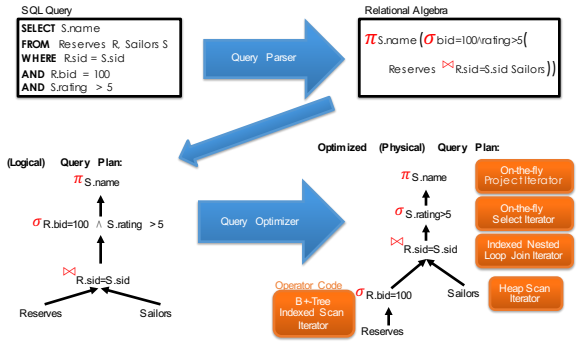
Architecture of a DBMS

Previously (2 Weeks)
How do we **store** and **access** data?

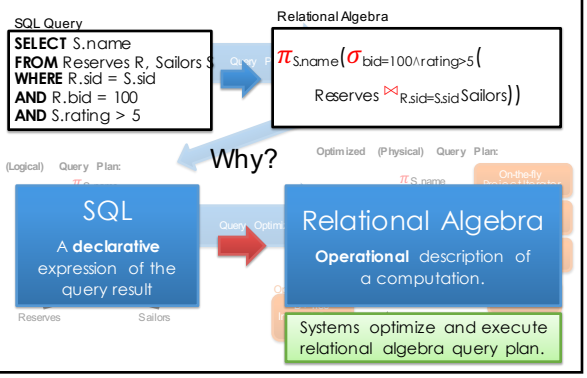
Today
How de we **represent** and **execute** computation?



Big Picture Overview



Big Picture Overview



Big Picture Overview

SQL Query

```
SELECT S.name
FROM Reserves R, Sailors S
WHERE R.sid = S.sid
AND R.bid = 100
AND S.rating > 5
```

Relational Algebra

$\pi_{S.name}(\sigma_{bid=100 \wedge rating>5}($ Reserves $\bowtie_{R.sid=S.sid}$ Sailors$))$

Why?

SQL
A **declarative** expression of the query result

Relational Algebra
Operational description of a computation.

Systems optimize and execute relational algebra query plan.

SQL (Structured Query Language)

- One of most **widely used** languages
- Introduced in **1970s**
  - IBM System R
- Key **System** Features: *Why do we like SQL*
  - Declarative:
    - Say ***what*** you want, not ***how*** to get it
    - Enables system to optimize the ***how***
  - Limited DSL → formal reasoning and optimization
- Foundation in formal Query Languages
  - **Relational Calculus**

```
SELECT S.name
FROM Reserves R, Sailors S
WHERE R.sid = S.sid
AND R.bid = 100
AND S.rating > 5
```

Formal Relational QL's

- **Relational Calculus:** (Basis for SQL)
  - Describe the result of computation
  - Based on first order logic
  - Tuple Relational Calculus **(TRC)**
    - $\{S \mid S \in Sailors \exists R \in Reserves (R.sid = S.sid \wedge R.bid = 103)\}$

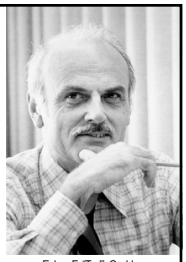
Are these equivalent?

Can we go from one to the other?

- **Relational Algebra:**
  - Algebra on sets
  - Operational description of transformations

Codd's Theorem

- Established equivalence in expressivity between :
  - **Relational Calculus***
  - **Relational Algebra**



Edgar F. "Ted" Codd
(1923 - 2003)
Turing Award 1981

- Why an import result?
  - Connects **declarative** representation of queries with **operational description**
  - **Constructive:** we can compile SQL into relational algebra

*Domain Independent Relational Calculus

Image obtained from https://en.wikipedia.org/wiki/Edgar_F._Codd

Big Picture Overview

SQL Query

```
SELECT S.name
FROM Reserves R, Sailors S
WHERE R.sid = S.sid
AND R.bid = 100
AND S.rating > 5
```

Relational Algebra

$\pi_{S.name}(\sigma_{bid=100 \wedge rating>5}($ Reserves $\bowtie_{R.sid=S.sid}$ Sailors$))$

Why?

SQL
A **declarative** expression of the query result

Relational Algebra
Operational description of a computation.

Systems optimize and execute relational algebra query plan.

Relational Algebra, Iterators, & Operator Implementations

R & G, Chapters
- Relational Algebra: 4.1 - 4.2
- Relational → Query Plan: 12
- Select, Project, **Join** operators: 14

- Day 2 (2/18/2016)

Berkeley cs186

Relational Algebra Preliminaries

- Algebra of *operators* on *relational instances*

$$\pi_{S.sname}(\sigma_{R.bid=100 \wedge S.rating>5}(R \bowtie_{R.sid=S.sid} S))$$

  - **Closed**: result is also a relational instance
    - Enables rich composition!
  - **Typed**: input schema determines output
  - Why is this important?

- Pure relational algebra has ***set* semantics**
  - **No duplicate** tuples in a relation instance
  - vs. SQL, which has *multiset* (bag) semantics
  - We will relax this in the system discussion

Relational Algebra Operators

Unary Operators: *operate on **single** relation instance*

- **Projection (π):** *Retains only desired columns* (vertical)
- **Selection (σ):** *Selects a subset of rows (horizontal)*
- **Renaming (ρ):** *Rename attributes and relations.*

Binary Operators: *operate on **pairs** of relation instances*

- **Union ($\cup$):** *Tuples in r1 or in r2.*
- **Intersection ($\cap$):** *Tuples in r1 and in r2.*
- **Set-difference ($-$):** *Tuples in r1, but not in r2.*
- **Cross-product ($\times$):** *Allows us to combine two relations.*
- **Joins ($\bowtie_\theta$, $\bowtie$):** Combine relations that satisfy predicates

Relational Algebra Operators

Unary Operators: *operate on **single** relation instance*

- **Projection (π):** *Retains only desired columns* (vertical)
- **Selection (σ):** *Selects a subset of rows (horizontal)*
- **Renaming (ρ):** *Rename attributes and relations.*

Binary Operators: *operate on **pairs** of relation instances*

- **Union ($\cup$):** *Tuples in r1 or in r2.*
- **Set-difference ($-$):** *Tuples in r1, but not in r2.*
- **Cross-product ($\times$):** *Allows us to combine two relations.*

Compound Operators: *operators built from earlier operators*

- **Intersection ($\cap$):** *Tuples in r1 and in r2.*
- **Joins ($\bowtie_\theta$, $\bowtie$):** Combine relations that satisfy predicates

Nautical Example Instances

Boats

bid	bname	color
101	Interlake	blue
102	Interlake	red
103	Clipper	green
104	Marine	red

R1

sid	bid	day
22	101	10/10/96
58	103	11/12/96

S1

sid	sname	rating	age
22	dustin	7	45.0
31	lubber	8	55.5
58	rusty	10	35.0

S2

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

Projection (π)

Selects a subset of columns (vertical)

$$\pi_{sname,rating}(S2)$$

List of Attributes

Relational *Instance* **S2**

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

sname	age
yuppy	35.0
lubber	55.5
guppy	35.0
rusty	35.0

- Corresponds to the **SELECT** list
- Schema determined by schema of attribute list
  - Names and types correspond to input attributes

Projection (π)

Selects a subset of columns (vertical)

$$\pi_{age}(S2)$$

Relational *Instance* **S2**

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

Multiset

age
35.0
55.5
35.0
35.0

Set

age
35.0
55.5

- Set semantics → results in fewer rows
  - Real systems don't automatically remove duplicates
  - why?

Selection (σ)

Selects a subset of rows (horizontal)

$$\sigma_{rating>8}(S2)$$

Selection Condition (Boolean Expression)

Relational *Instance* **S2**

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

sid	sname	rating	age
28	yuppy	9	35.0
58	rusty	10	35.0

- Corresponds to the **WHERE** clause
- Output schema same as input
- Duplicate Elimination?

Composing Select and Project

- Names of sailors with rating > 8

$$\pi_{name}(\sigma_{rating>8}(S2))$$

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

$\sigma_{rating>8}$

sid	sname	rating	age
28	yuppy	9	35.0
58	rusty	10	35.0

π_{name}

sname
yuppy
rusty

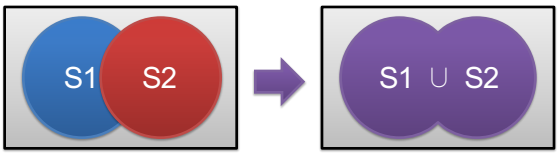
- What about:

$$\sigma_{rating>8}(\pi_{name}(S2))$$

Invalid types. Input to $\sigma_{rating>8}$ does not contain rating.

Union ($\cup$)

S1 $\cup$ S2



Two input relations, must be *compatible*:
- Same number of fields.
- "Corresponding" fields have same **type**
- SQL Expression: UNION

Union ($\cup$)

Relational *Instance* **S1**

sid	sname	rating	age
22	dustin	7	45.0
31	lubber	8	55.5
58	rusty	10	35.0

Relational *Instance* **S2**

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

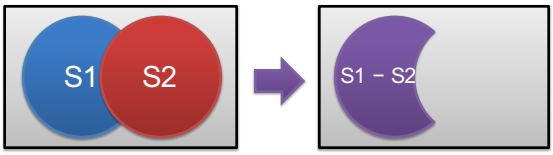
S1 $\cup$ S2

sid	sname	rating	age
22	dustin	7	45
28	yuppy	9	35.0
31	lubber	8	55.5
44	guppy	5	35.0
58	rusty	10	35.0

Duplicate elimination in practice?
- UNION vs UNION ALL?

Set Difference (–)

S1 – S2



Same as with union, both input relations must be *compatible*.

SQL Expression: EXCEPT

Set Difference (–)

Relational *Instance* **S1**

sid	sname	rating	age
22	dustin	7	45.0
31	lubber	8	55.5
58	rusty	10	35.0

Relational *Instance* **S2**

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

S1 – S2

sid	sname	rating	age
22	dustin	7	45

Symmetric?

S2 – S1

sid	sname	rating	age
28	yuppy	9	35.0
44	guppy	5	35.0

Duplicate elimination?
- Not required
- EXCEPT vs EXCEPT ALL

Monotonicity and Set Difference

- A relational op. Q is monotone in R if:

$R_1 \subseteq R_2 \Rightarrow Q(R_1, S, T, \ldots) \subseteq Q(R_2, S, T, \ldots)$

- For example UNION ($\cup$) is monotone:

$R_1 \subseteq R_2 \Rightarrow S \cup R_1 \subseteq S \cup R_2$

- Set difference?

Monotonicity and Set Difference

- A relational op. Q is monotone in R if:

$R_1 \subseteq R_2 \Rightarrow Q(R_1, S, T, \ldots) \subseteq Q(R_2, S, T, \ldots)$

- Set difference (−) where $S = \{a,b,c\}$:

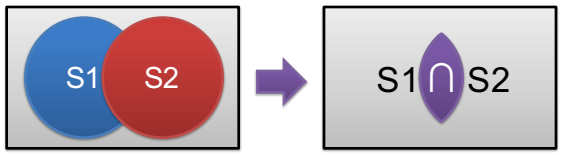
$R_1 = \{a\} \subseteq R_2 = \{a,b\} \overset{?}{\Rightarrow} S - \{a\} \subseteq S - \{a,b\}$

$\{b,c\} \not\subseteq \{c\}$

- *Implication*: set difference is **blocking**
  - For S – R, need to have full contents of R before emitting **any** results

Intersection ($\cap$)

S1 $\cap$ S2



Same as with union, both input relations must be *compatible*.

SQL Expression: INTERSECT

Intersection ($\cap$)

Relational *Instance* **S1**

sid	sname	rating	age
22	dustin	7	45.0
31	lubber	8	55.5
58	rusty	10	35.0

Relational *Instance* **S2**

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

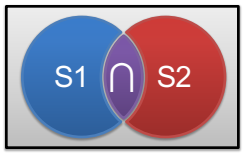
S1 $\cap$ S2

sid	sname	rating	age
31	lubber	8	55.5
58	rusty	10	35.0

Is intersection essential?

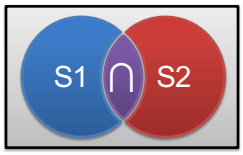
- Implement it with earlier ops. ?

Intersection ($\cap$)



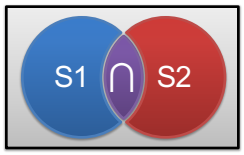
S1 $\cap$ S2 = ?

Intersection ($\cap$)

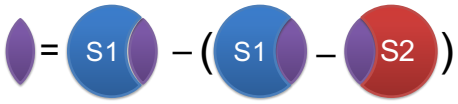


S1 $\cap$ S2 = S1 – ?

Intersection (∩)



$S1 \cap S2 = S1 - (S1 - S2)$



Is intersection monotonic?

$R_1 \subseteq R_2 \Rightarrow S \cap R_1 \subseteq S \cap R_2$

Yes!

Cross-Product (×)

R1 × S1: *Each row of **R1** paired with each row of **S1***

R1:

sid	bid	day
22	101	10/10/96
58	103	11/12/96

× **S1:**

sid	sname	rating	age
22	dustin	7	45.0
31	lubber	8	55.5
58	rusty	10	35.0

=

R1 × S1

sid	bid	day	sid	sname	rating	age
22	101	10/10/96	22	dustin	7	45.0
22	101	10/10/96	31	lubber	8	55.5
22	101	10/10/96	58	rusty	10	35.0
58	103	11/12/96	22	dustin	7	45.0
58	103	11/12/96	31	lubber	8	55.5
58	103	11/12/96	58	rusty	10	35.0

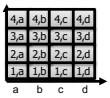
How many rows in the result? |R1| * |R2|

Schema compatibility? No requirements.

One field per field in original schemas.

What about duplicate names? Renaming operator

Sometimes also called **Cartesian Product:**



Renaming (ρ = *"rho"*)

Renames relations and their attributes:

$\rho(\text{Temp1}(1 \rightarrow \text{sid1}, 4 \rightarrow \text{sid2}), \text{R1} \times \text{S1})$

Output Relation Name | Renaming List position → New Name | Input Relation

R1 × S1

sid	bid	day	sid	sname	rating	age
22	101	10/10/96	22	dustin	7	45.0
22	101	10/10/96	31	lubber	8	55.5
22	101	10/10/96	58	rusty	10	35.0
58	103	11/12/96	22	dustin	7	45.0
58	103	11/12/96	31	lubber	8	55.5
58	103	11/12/96	58	rusty	10	35.0

Temp1

sid1	bid	day	sid2	sname	rating	age
22	101	10/10/96	22	dustin	7	45.0
22	101	10/10/96	31	lubber	8	55.5
22	101	10/10/96	58	rusty	10	35.0
58	103	11/12/96	22	dustin	7	45.0
58	103	11/12/96	31	lubber	8	55.5
58	103	11/12/96	58	rusty	10	35.0

- Note that relational algebra doesn't require names.
  - We could just use positional arguments. $\pi_{f5}(\sigma_{f6>f8}(S2))$
- Difficult to read …

Compound Operator: Join

- Joins are compound operators (like intersection):
  - Built using **cross product** & **selection** and sometimes **projection** (for natural join)
- Hierarchy of common kinds:
  - **Theta Join ($\bowtie_\theta$):** *join on logical expression θ*
    - **Equi-Join:** *theta join with conjunction equalities*
      - **Natural Join ($\bowtie$):** *equi-join on all matching column names*
- Note: we should use a good join algorithm, not a cross-product if we can avoid it!!

Theta Join ($\bowtie_\theta$)

$R \bowtie_\theta S = \sigma_\theta(R \times S)$

Example: *More senior sailors for each sailor.*

$S1 \bowtie_{age < age} S1$

S1:

sid	sname	rating	age
22	dustin	7	45.0
31	lubber	8	55.5
58	rusty	10	35.0

This predicate doesn't make any sense!

Could switch to positional arguments

Theta Join ($\bowtie_\theta$)

$R \bowtie_\theta S = \sigma_\theta(R \times S)$

Example: *More senior sailors for each sailor.*

$S1 \bowtie_{f4 < f8} S1$

S1:

f1	f2	f3	f4
22	dustin	7	45.0
31	lubber	8	55.5
58	rusty	10	35.0

Difficult to read
→ use ρ operator

S1 × S1

S1				S1			
f1	f2	f3	f4	f5	f6	f7	f8
22	dustin	7	45.0	22	dustin	7	45.0
22	dustin	7	45.0	31	lubber	8	55.5
22	dustin	7	45.0	58	rusty	10	35.0
31	lubber	8	55.5	22	dustin	7	45.0
31	lubber	8	55.5	31	lubber	8	55.5
31	lubber	8	55.5	58	rusty	10	35.0
58	rusty	10	35.0	22	dustin	7	45.0
58	rusty	10	35.0	31	lubber	8	55.5
58	rusty	10	35.0	58	rusty	10	35.0

Theta Join ($\bowtie_\theta$)

$$R \bowtie_\theta S = \sigma_\theta(R \times S)$$

Example: *More senior sailors for each sailor.*

$S1 \bowtie_{age < age2} \rho((8 \rightarrow age2), S1)$

S1:

sid	sname	rating	age
22	dustin	7	45.0
31	lubber	8	55.5
58	rusty	10	35.0

S1 × S1

S1				S1			
sid	sname	rating	age	sid	sname	rating	age2
22	dustin	7	45.0	22	dustin	7	45.0
22	dustin	7	45.0	31	lubber	8	55.5
22	dustin	7	45.0	58	rusty	10	35.0
31	lubber	8	55.5	22	dustin	7	45.0
31	lubber	8	55.5	31	lubber	8	55.5
31	lubber	8	55.5	58	rusty	10	35.0
58	rusty	10	35.0	22	dustin	7	45.0
58	rusty	10	35.0	31	lubber	8	55.5
58	rusty	10	35.0	58	rusty	10	35.0

Theta Join ($\bowtie_\theta$)

$$R \bowtie_\theta S = \sigma_\theta(R \times S)$$

Example: *More senior sailors for each sailor.*

$S1 \bowtie_{age < age2} \rho((8 \rightarrow age2), S1)$

S1:

sid	sname	rating	age
22	dustin	7	45.0
31	lubber	8	55.5
58	rusty	10	35.0

S1				S1			
sid	sname	rating	age	sid	sname	rating	age2
22	dustin	7	45.0	31	lubber	8	55.5
58	rusty	10	35.0	22	dustin	7	45.0
58	rusty	10	35.0	31	lubber	8	55.5

- *Result schema* same as that of cross-product.
- Special Case:
  - **Equi-Join:** *theta join with conjunction equalities*
    - *Special special case* ***Natural Join*** …

Natural Join ($\bowtie$)

Special case of ***equi-join*** *in which equalities are specified for all matching fields and duplicate fields are projected away*

$$R \bowtie S = \pi_{\text{unique fld.}}\, \sigma_{\text{eq. matching fld.}}(R \times S)$$

- Compute $R \times S$
- **Select** rows where fields appearing in both relations have equal values
- **Project** onto the set of all unique fields.

Natural Join ($\bowtie$)

$$R \bowtie S = \pi_{\text{unique fld.}}\, \sigma_{\text{eq. matching fld.}}(R \times S)$$

Example:

R1:

sid	bid	day
22	101	10/10/96
58	103	11/12/96

S1:

sid	sname	rating	age
22	dustin	7	45.0
31	lubber	8	55.5
58	rusty	10	35.0

$R1 \bowtie S1$

sid	bid	day	sid	sname	rating	age
22	101	10/10/96	22	dustin	7	45.0
22	101	10/10/96	31	lubber	8	55.5
22	101	10/10/96	58	rusty	10	35.0
58	103	11/12/96	22	dustin	7	45.0
58	103	11/12/96	31	lubber	8	55.5
58	103	11/12/96	58	rusty	10	35.0

Natural Join ($\bowtie$)

$$R \bowtie S = \pi_{\text{unique fld.}}\, \sigma_{\text{eq. matching fld.}}(R \times S)$$

Example:

R1:

sid	bid	day
22	101	10/10/96
58	103	11/12/96

S1:

sid	sname	rating	age
22	dustin	7	45.0
31	lubber	8	55.5
58	rusty	10	35.0

$R1 \bowtie S1$

sid	bid	day	sname	rating	age
22	101	10/10/96	dustin	7	45.0
58	103	11/12/96	rusty	10	35.0

Commonly used for foreign key joins (as above).

Exercise:

Boats(bid, bname, color)
Sailors(sid, sname, rating, age)
Reserves(sid, bid, day)

Find names of sailors who've reserved boat #103

- Solution 1:

$$\pi_{\text{sname}}(\sigma_{\text{bid}=103}(\text{Sailors} \bowtie \text{Reserves}))$$

- Solution 2:

$$\pi_{\text{sname}}(\text{Sailors} \bowtie \sigma_{\text{bid}=103}(\text{Reserves}))$$

Exercise:

Boats(bid, bname, color)
Sailors(sid, sname, rating, age)
Res(sid, bid, day)

Find names of sailors who've reserved a red boat

- Solution 1:

$$\pi_{sname}(\sigma_{color='red'}(Boats) \bowtie Res \bowtie Sailors)$$

- More "efficient" Solution 2:

$$\pi_{sname}(\pi_{sid}(\pi_{bid}(\sigma_{color='red'}(Boats)) \bowtie Res) \bowtie Sailors)$$

In general many possible equivalent expressions: **algebra**…

Relational Algebra Rules

- **Operator Precedence:**
  - Unary operators before binary operators
- **Selections:**
  - $\sigma_{c1 \wedge \ldots \wedge cn}(R) \equiv \sigma_{c1}(\ldots(\sigma_{cn}(R))\ldots)$ (*cascade*)
  - $\sigma_{c1}(\sigma_{c2}(R)) \equiv \sigma_{c2}(\sigma_{c1}(R))$ (*commute*)
- **Projections:**
  - $\pi_{a1}(R) \equiv \pi_{a1}(\ldots(\pi_{a1, \ldots, an-1}(R))\ldots)$ (*cascade*)
- **Cartesian Product**
  - $R \times (S \times T) \equiv (R \times S) \times T$ (associative)
  - $R \times S \equiv S \times R$ (commutative)
  - *Applies for joins as well but* be careful with join predicates …

Caution with Natural Join

Boats(bid, bname, color)
Sailors(sid, sname, rating, age)
Reserves(sid, bid, day)

- Consider the following:

$$Boats \bowtie_{bid} Reserves \bowtie_{sid} Sailors$$

- Commute and Associate:

$$Boats \bowtie_{bid} (Sailors \bowtie_{sid} Reserves)$$

  - Incompatible join predicate:

$$(Boats \bowtie_{bid} Sailors) \bowtie_{sid} Reserves$$

Caution with Natural Join

Boats(bid, bname, color)
Sailors(sid, sname, rating, age)
Reserves(sid, bid, day)

- Consider the following:

$$Boats \bowtie_{bid} Reserves \bowtie_{sid} Sailors$$

- Commute and Associate:

$$Boats \bowtie_{bid} (Sailors \bowtie_{sid} Reserves)$$

  - Incompatible join predicate:

$$(Boats \times Sailors) \bowtie_{sid,bid} Reserves$$

Relational Algebra Rules

Commuting of selection operators

- $\sigma_c(R \times S) \equiv \sigma_c(R) \times S$ (c only has fields in R)
- $\sigma_c(R \bowtie S) \equiv \sigma_c(R) \bowtie S$ (c only has fields in R)

Commuting of projection operators

- $\pi_a(R \times S) \equiv \pi_{a1}(R) \times \pi_{a2}(S)$
  - a_1 is subset of a that mentions R and a_2 is subset of a that mentions S
  - Similar result holds for joins

Division (/) Compound Operator

"Find the names of sailors that reserved all boats."

- Consider Relations: A(x,y) and B(y)
  - $A / B = \{ x \mid \forall y \in B \ ((x,y) \in A) \}$
  - all entries **x** in **A** such that for all **y** in **B** there is an **(x,y)** in **A**
- Pictorially:

A	
x	y
x_1	y_1
x_1	y_2
x_2	y_1
x_3	y_2

/

B
y
y_1
y_2

=

A / B
x
x_1

How do we implement division in relational algebra?

Division (/) Compound Operator

- If all the **x** values in **A** are in the result what would **A** look like?

$\pi_x(A) \times B =$

x	y
x_1	y_1
x_1	y_2
x_2	y_1
x_2	y_2
x_3	y_1
x_3	y_2

A

x	y
x_1	y_1
x_1	y_2
x_2	y_1
x_3	y_2

B

y
y_1
y_2

Division (/) Compound Operator

- If all the **x** values in **A** were in the result what would **A** look like?

$\pi_x(A) \times B$

- Which of these tuples are we missing from **A**:

A

x	y
x_1	y_1
x_1	y_2
x_2	y_1
x_3	y_2

B

y
y_1
y_2

$(\pi_x(A) \times B) - A$ = *Missing (x,y) values*

x	y
x_2	y_2
x_3	y_1

Division (/) Compound Operator

- If all the **x** values in **A** were in the result what would **A** look like?

$\pi_x(A) \times B$

- Which of these tuples are we missing from **A** and what are their **x** values:

A

x	y
x_1	y_1
x_1	y_2
x_2	y_1
x_3	y_2

B

y
y_1
y_2

$\pi_x((\pi_x(A) \times B) - A)$ = *Disqualified x values*

x
x_2
x_3

Division (/) Compound Operator

- If all the **x** values in **A** were in the result what would **A** look like?

$\pi_x(A) \times B$

- Which of these tuples are we missing from **A** and what are their **x** values:

A

x	y
x_1	y_1
x_1	y_2
x_2	y_1
x_3	y_2

B

y
y_1
y_2

$\pi_x((\pi_x(A) \times B) - A)$

- Remove disqualified **x** values:

$A / B = \pi_x(A) - \pi_x((\pi_x(A) \times B) - A)$

Operators in Extended Algebra

- Group By / Aggregation Operator (γ):

$\gamma_{age,\ AVG(rating)}(Sailors)$

– *With selection (HAVING clause):*

$\gamma_{age,\ AVG(rating),\ COUNT(*)>2}(Sailors)$

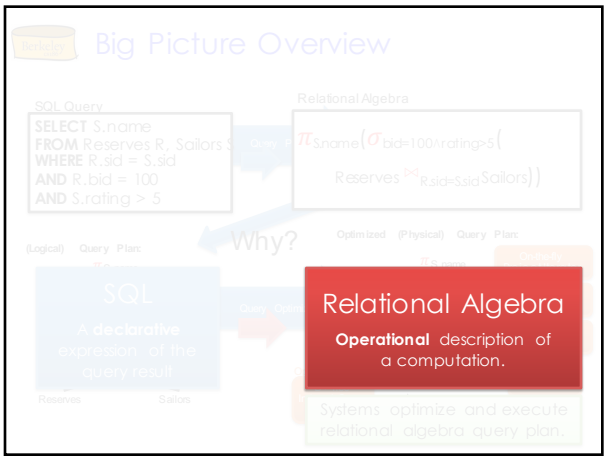
- Textbook uses two operators:

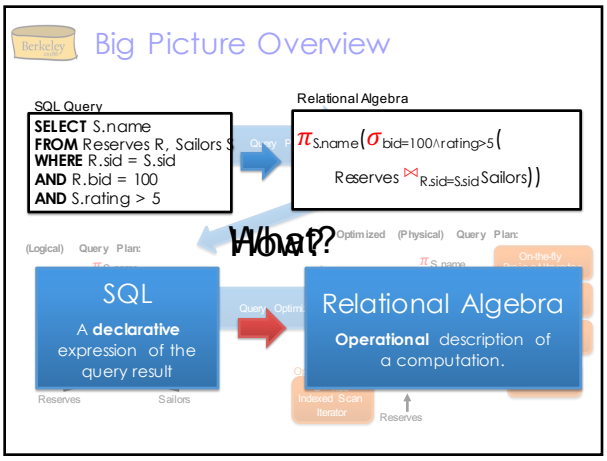
$GROUP\ BY_{age,\ AVG(rating)}(Sailors)$

$HAVING_{COUNT(*)>2}(GROUP\ BY_{age,\ AVG(rating)}(Sailors))$

Summary

- Relational Algebra: a small set of operators mapping relations to relations
  - Operational, in the sense that you specify the explicit order of operations
  - A closed set of operators! Mix and match.
- Basic ops include: $\sigma, \pi, \times, \cup, -$
- Important compound ops: $\cap, \bowtie$





Translating SQL to Relational Algebra

Restrict attention to single ($\sigma\pi\times$- query)

```
SELECT  S.sid, MIN (R.day)
FROM  Sailors S, Reserves R, Boats B
WHERE  S.sid = R.sid AND R.bid = B.bid AND B.color = "red"
GROUP BY S.sid
HAVING  COUNT (*) >= 2
```

Select-Project-Join Query

For each sailor with at least two reservations for red boats, find the sailor id and the earliest date on which the sailor has a reservation for a red boat.

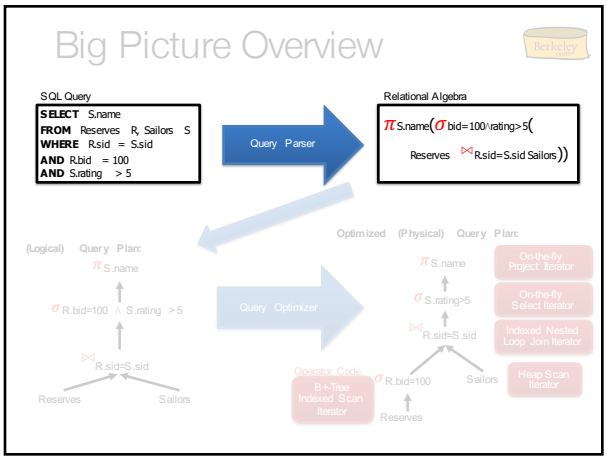
Translating SQL to Relational Algebra

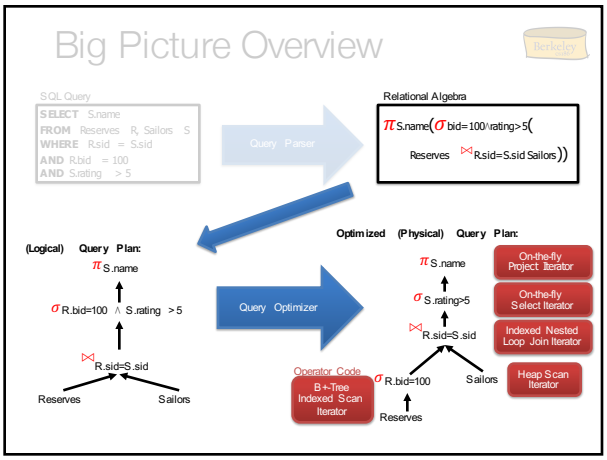
Restrict attention to single ($\sigma\pi\times$- query)

```
SELECT  S.sid, MIN (R.day)
FROM  Sailors S, Reserves R, Boats B
WHERE  S.sid = R.sid AND R.bid = B.bid AND B.color = "red"
GROUP BY S.sid
HAVING  COUNT (*) >= 2
```

Select-Project-Join Query

$$\pi_{\text{S.sid, MIN(R.day)}}\Big(\text{HAVING}_{\text{COUNT(*)}>2}\big(\text{GROUP BY}_{\text{S.Sid}}\big(\sigma_{\text{B.color = "red"}}\big(\text{Sailors} \bowtie \text{Reserves} \bowtie \text{Boats}\big)\big)\big)\Big)$$

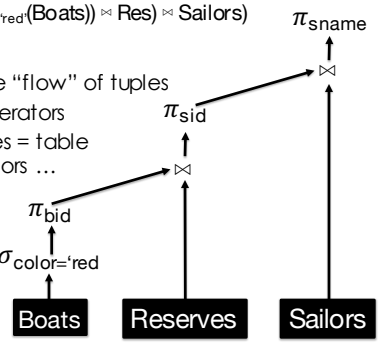




Relational Operators as a Graph

$\pi_{sname}(\pi_{sid}(\pi_{bid}(\sigma_{color='red'}(Boats)) \bowtie Res) \bowtie Sailors)$

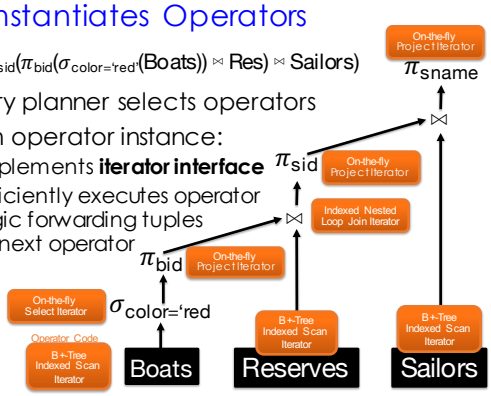
- Query plan
  - Edges encode "flow" of tuples
  - Vertices = Operators
  - Source vertices = table access operators …
- Also called dataflow graph



Query Executor Instantiates Operators

$\pi_{sname}(\pi_{sid}(\pi_{bid}(\sigma_{color='red'}(Boats)) \bowtie Res) \bowtie Sailors)$

- Query planner selects operators
- Each operator instance:
  - Implements **iterator interface**
  - Efficiently executes operator logic forwarding tuples to next operator



Iterators Interface

The relational operators are all subclasses of the class iterator:

```
abstract class iterator {
        // Invoked when "wiring" dataflow
        void setup(List<Iterator> inputs, args);
        void init(args);   // Invoked before calling next
        tuple next();  // Invoked repeatedly
        void close();  // Invoked when finished
    }
```

Note:

- **Pull:** based computation model
  - e.g., Console calls **init** and **next** which propagates down graph
  - **Blocking:** *init* and *next* don't return until done.
    - **Asynchronous** models more complex but could be more efficient
- **Encapsulation**: any iterator can be input to any other!
- **State:** iterators may maintain substantial state
  - e.g., hash tables, running counts, large sorted files …

Example: Scan Heap File

```
class heap_scan_iterator extends iterator {
   void init() {
     current_page = file_manager.begin(heap_file, 'MRU')
     pageIter = current_page.iterator()
   }
   tuple next() { // Invoked repeatedly
     if (!pageIter.hasNext()) {
       current_page = file_manager.next(current_page);
       pageIter = current_page.iterator();
     }
     if current_page == null { return EOF; }
     else { return pageIter.next(); }
   }
   void close() { // Invoked when finished
       file_manager.close(heap_file);
   }
}
```

pseudocode … might have pseudobugs

Example: Select

```
class select_iterator extends iterator {
   void setup(List<Iterator> inputs, predicate) …

   void init() { input[0].init(); }

   tuple next() { // Invoked repeatedly
       res = input[0].next();
       while( res not EOF && not predicate(res) ) {
         res = input[0].next();
       }
       return res;
   }

   void close() { // Invoked when finished
       input[0].close();
   }
}
```

pseudocode … might have pseudobugs

Selective Access Paths

- **Access Path:** How data is accessed from input relations:
  - File scan
  - Indexed lookup: *Attr* **op** *Value*
- **Selectivity of Access Path:** Number of pages retrieved
- **Goal:** *Use the most selective access path*

Example: B+-Tree (by Ref Clustered)

```
class index_select_iterator extends iterator {
  void init(keyRange) { // might be invoked repeatedly
    index = file_manager.getIndex(file)
    (leaf_page, offset) = index.findBegin(keyRange)
  }
  tuple next() { // Invoked repeatedly
    if (offset == END_OF_PAGE) {
      (leaf_page, offset) = index.next_leaf(leaf_page)
    }
    (k, rid) = leaf_page.read(offset)
    if (rid == EOF || k not in keyRange) { return EOF; }
    return file_manager.get_record(rid);
  }
  void close() { // Invoked when finished
      file_manager.close(index);
  }
}
```

pseudocode … might have pseudobugs

Example: B+-Tree (by Ref Unclustered)

```
class index_select_iterator extends iterator {
  void init(keyRange) { // might be invoked repeatedly
    index = file_manager.getIndex(file)
    tmpFile = index.scanRecordIdsToTmpFile(keyRange)
    iter = new sort_iterator(tmpFile)
    iter.init()
  }
  tuple next() { // Invoked repeatedly
    rid = iter.next()
    if (rid == EOF) { return EOF; }
    return file_manager.get_record(rid);
  }
  void close() { // Invoked when finished
    file_manager.close(index);
    iter.close()
    file_manager.destroy(tmpFile);
  }
```

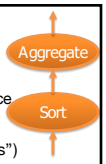
pseudocode … might have pseudobugs

Example: Sort

- init():
  - generate the sorted runs on disk
  - Open each sorted run file
  - load (sorted) heap with first tuple in each run
- next():
  - Get top tuple from heap
  - Load next tuple from file which top record came from and place in heap
  - return tuple (or EOF -- "End of Fun" -- if no tuples remain)
- close():
  - deallocate the runs files

Get to do this in the homework but with an additional distributed partitioning step!

Example: Sort → Group By



- The Sort iterator ensures all its tuples are output in sequence
- The Aggregate iterator keeps running info ("transition values") on agg functions in the SELECT list, per group
  - E.g., for COUNT, it keeps `count-so-far`
  - For SUM, it keeps `sum-so-far`
  - For AVG it keeps `sum-so-far` *and* `count-so-far`
- As soon as the Aggregate iterator sees a tuple from a new group:
  1. It produces output for the old group based on agg function
     E.g. for AVG it returns (`sum-so-far/count-so-far`)
  2. It resets its running info.
  3. It updates the running info with the new tuple's info

Example: Hash Group By

You get to do this in your homework!

Iterators in PySpark (Homework)


```
def init(iterA, iterB):
    a = list(iterA)
    b = list(iterB)
    return iter(zip(a, b))

rdd = sc.parallelize(range(10), 2)

print rdd.zipPartitions(rdd, init).collect()
```

Uses a lot of memory (avoid "materializing" partition)

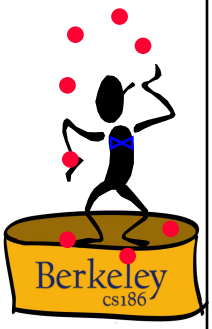
```
[(0, 0), (1, 1), (2, 2), (3, 3), (4, 4), (5, 5),
 (6, 6), (7, 7), (8, 8), (9, 9)]
```

Iterators in PySpark (Homework)

```
def init(iterA, iterB):
    for a in iterA:
        yield (a, iterB.next())

rdd = sc.parallelize(range(10), 2)

print rdd.zipPartitions(rdd,  init).collect()

 [(0, 0), (1, 1), (2, 2), (3, 3), (4, 4), (5, 5),
  (6, 6), (7, 7), (8, 8), (9, 9)]
```

Join Operators

R&G 14.4



Joins

$$R \bowtie_\theta S = \sigma_\theta(R \times S)$$

- Joins are very common
- R × S is large →
  - Minimize IO
  - Exploit θ to reduce computation (selectivity)
- Join techniques we will cover today:
  - Nested-loops join
  - Index-nested loops join
  - Sort-merge join
  - Hash Joins

Schema for Examples

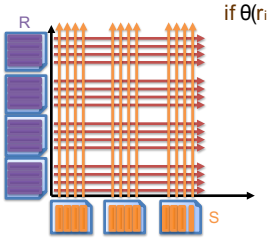
Cost Notation
- [R] : the number of **pages** to store R
- p_R : number of **records per page** of R
- |R| : the **cardinality** (*number of records*) of R
  - $|R| = p_R * [R]$

Reserves (*sid*: int, *bid*: int, *day*: date, *rname*: string)
- [R]=1000, p_R=100, |R| = 100,000

Sailors (*sid*: int, *sname*: string, *rating*: int, *age*: real)
- [S]=500, p_S=80, |S| = 40,000

Simple Nested Loops Join

R ⋈ S: foreach record r in R do
  foreach record s in S do
    if θ(r_i, s_j) then add <r, s> to result



Cost?
- $([R] * p_R) * [S] + [R]$
- 100,000 * 500 + 1,000
  - 50,001,000

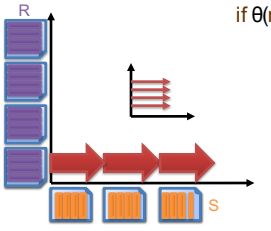
Swap R and S?
- $([S] * p_S) * [R] + [S]$
- 40,000 * 1,000 + 500
  - 40,000,500
  - @10ms → 4.63 Days!

- Can we do better?

[R]=1000, p_R=100, |R| = 100,000
[S]=500, p_S=80, |S| = 40,000

Page-Oriented Nested Loop Join

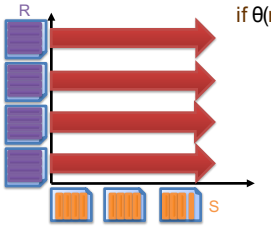
R ⋈ S: foreach page b_R in R do
  foreach page b_S in S do
    foreach record r in b_R do
      foreach record s in b_S do
        if θ(r_i,s_j) then add <r, s> to result



[R]=1000, p_R=100, |R| = 100,000
[S]=500, p_S=80, |S| = 40,000

Page-Oriented Nested Loop Join

R ⋈ S:
foreach page b_R in R do
  foreach page b_S in S do
    foreach record r in b_R do
      foreach record s in $b_S$do
        if θ(r_i,s_j) then add <r, s> to result



Cost?
- [R] * [S] + [R]
- 1,000 * 500 + 1,000
  - 501,000

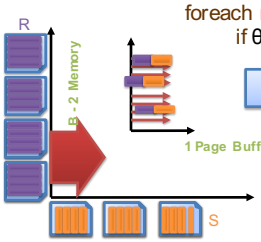
Swap R and S?
- [S] * [R] + [S]
- 500 * 1,000 + 500
  - 500,500
  - @10ms → 1.39 hours!
  - Can we do better?

[R]=1000, p_R=100, |R| = 100,000
[S]=500, p_S=80, |S| = 40,000

Block Nested Loop Join

Assume B Pages of Memory

R ⋈ S:
foreach **block** b_R of **B-2 pages** in R do
  foreach page b_S in S do
    foreach record r in b_R do
      foreach record s in $b_S$do
        if θ(r_i,s_j) then add <r, s> to result

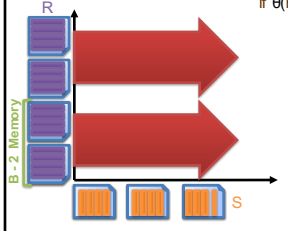


[R]=1000, p_R=100, |R| = 100,000
[S]=500, p_S=80, |S| = 40,000

Block Nested Loop Join

Assume B Pages of Memory

R ⋈ S:
foreach **block** b_R of **B-2 pages** in R do
  foreach page b_S in S do
    foreach record r in b_R do
      foreach record s in $b_S$do
        if θ(r_i,s_j) then add <r, s> to result



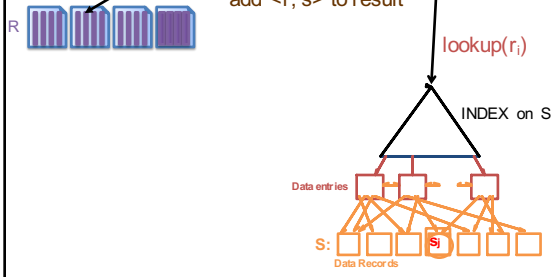
Cost: (Assume B = 102)
- ([R]/(B-2)) * [S] + [R]
- 10 * 500 + 1,000
  - 6,000

Swap R and S?
- ([S] /(B-2)) * [R] + [S]
- 5 * 1,000 + 500
  - 5,500
  - @10ms → 55 seconds!
  - Can we do better?

[R]=1000, p_R=100, |R| = 100,000
[S]=500, p_S=80, |S| = 40,000

Index Nested Loops Join

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



Index Nested Loops Join

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

Cost = [R] + ([R] * p_R) * cost to find matching S tuples

- If index uses Alt. 1 → cost to traverse tree from root to leaf. (e.g., *2-4 IOs*)
- For Alt. 2 or 3:
  1. Cost to lookup RID(s); typically 2-4 IOs for B+Tree.
  2. Cost to retrieve records from RID(s); depends on clustering.
     - **Clustered index**: 1 I/O per page of matching S tuples.
     - **Unclustered**: up to 1 I/O per matching S tuple (w/o sorting)

Index Nested Loops Join

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

Cost = [R] + ([R] * p_R) * cost to find matching S tuples

- Assume alternative 2 (by reference) unclustered
  - 1 matching tuple (e.g., primary key lookup)
  - only leaves and data entries not in cache → 2 IOs
- Cost:
  - R⋈S: 1000 + (1,000*100)*2 = 201,000
  - S⋈R: 500 + (500*80)*2 = 80,500 → 13 Minutes
  - **Block Nested Loop**: 5,500
    - 55 Seconds …

Is the 1 matching tuple assumption reasonable?

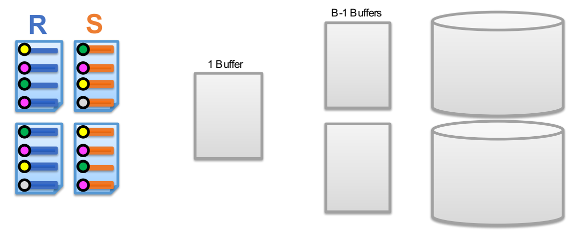
[R]=1,000, p_R=100, |R| = 100,000
[S]=500, p_S=80, |S| = 40,000

Grace Hash Join

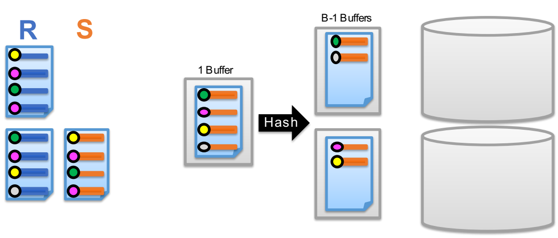
$R \bowtie_\theta S = \sigma_\theta(R \times S)$

- Requires equality predicate θ:
  - Works for **Equi-Joins** & **Natural Joins**
- Two Stages:
  - Divide – **Partition** tuples from R and S by join key
    - all tuples for a given key in same partition
  - Conquer – **Build & Probe** a separate hash table for each partition
    - Assume **partition** of smaller rel. fits in memory
      - Recurse if necessary…

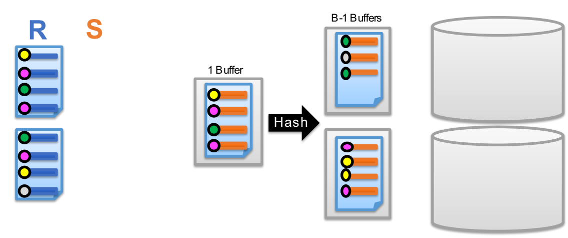
Grace Hash Join: *Partition*



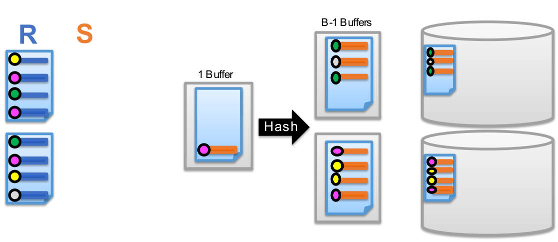
Grace Hash Join: *Partition*



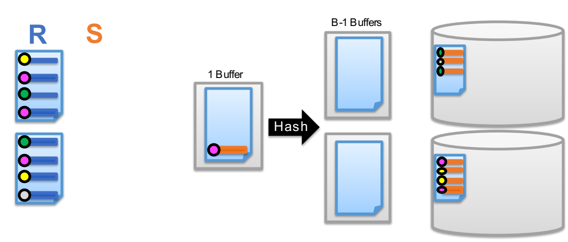
Grace Hash Join: *Partition*



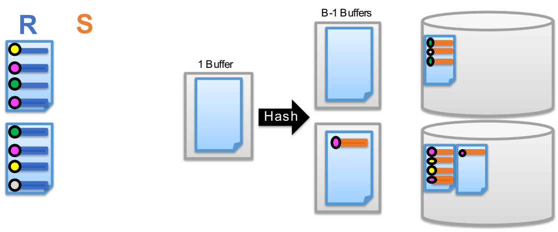
Grace Hash Join: *Partition*



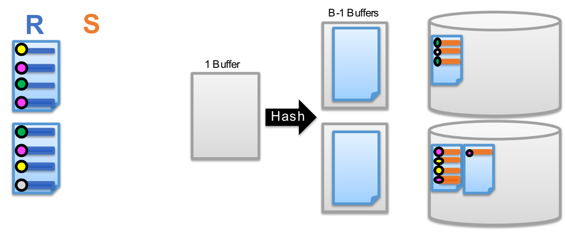
Grace Hash Join: *Partition*



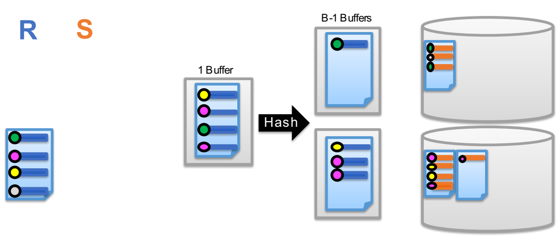
Grace Hash Join: *Partition*



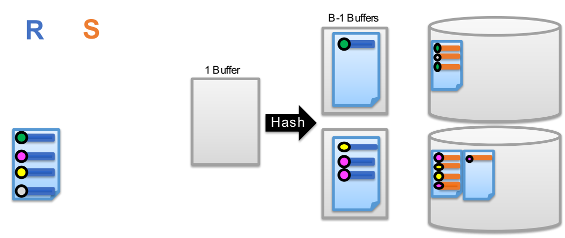
Grace Hash Join: *Partition*



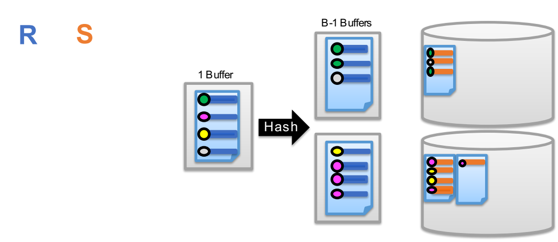
Grace Hash Join: *Partition*



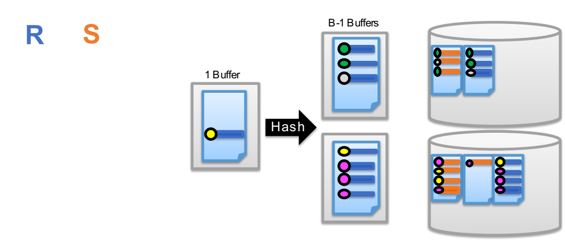
Grace Hash Join: *Partition*



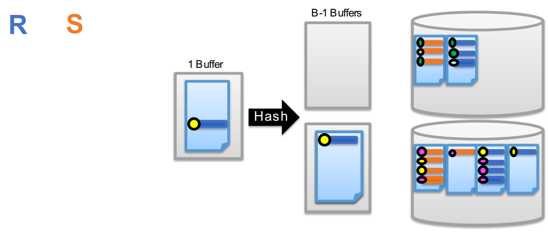
Grace Hash Join: *Partition*



Grace Hash Join: *Partition*

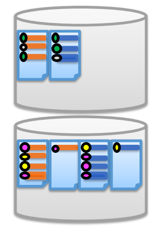


Grace Hash Join: ***Partition***

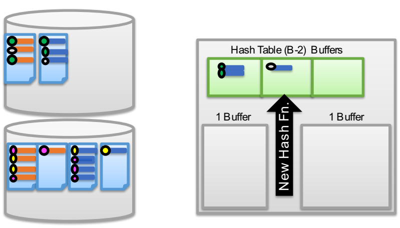


Grace Hash Join: ***Partition***

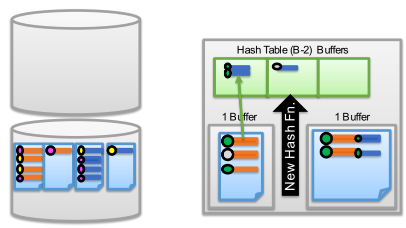
- Each key is assigned to one partition
  – e.g., green keys only here →
- Sensitive to key Skew
  – Fuchsia Key
- Each partition could be different machine
  – PySpark Homework …



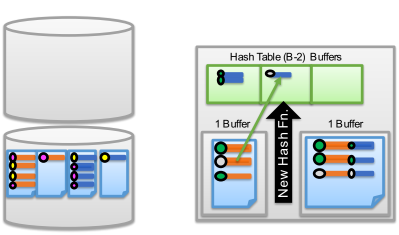
Grace Hash Join: ***Build*** ***& Probe***



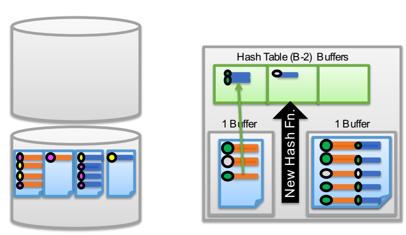
Grace Hash Join: ***Build & Probe***



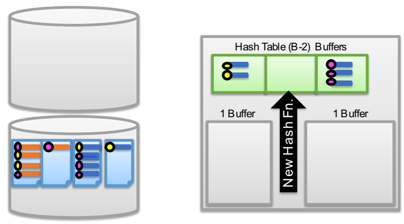
Grace Hash Join: ***Build & Probe***



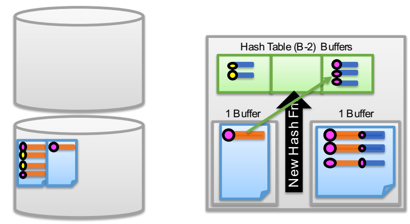
Grace Hash Join: ***Build & Probe***



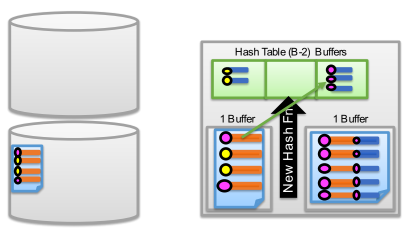
Grace Hash Join: ***Build*** & ***Probe***



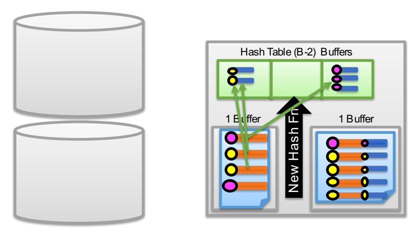
Grace Hash Join: ***Build*** & ***Probe***



Grace Hash Join: ***Build*** & ***Probe***



Grace Hash Join: ***Build*** & ***Probe***



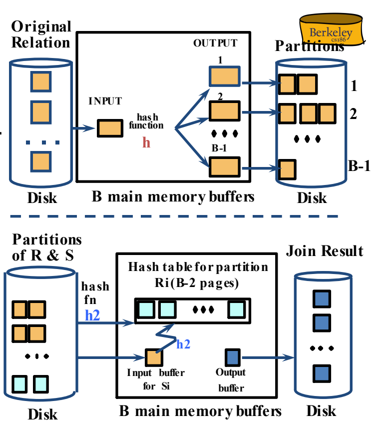
You get the idea …

Hash Join

Do this for R.
Then do it for S.

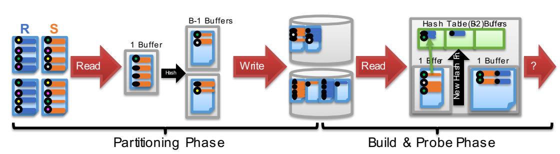
Build an R H.T.

Stream S & *probe*.



Cost of Hash Join

[R]=1000, p_R=100, |R| = 100,000
[S]=500, p_S=80, |S| = 40,000



- Partitioning phase: read+write both relations
  - ⇒ 2([R]+[S]) I/Os
- Matching phase: read both relations, forward output
  - ⇒ [R]+[S]
- Total cost of 2-pass hash join = 3([R]+[S])
  - 3 * (1000 + 500) = 4500
  - 45 seconds @10ms per page, probably **much** faster, why?
    - Serial IO so no seek overhead → 10ms is much too high

Mechanisms of Rendezvous

Hashing & Sorting

Sort-Merge Join

$R \bowtie_\theta S = \sigma_\theta(R \times S)$

- Requires equality predicate θ:
  - Works for **Equi-Joins** & **Natural Joins**
- Two Stages:
  - **Sort** tuples in R and S by join key
    - all tuples with same key in consecutive order
    - input might already sorted … why?
  - **Merge** scan the sorted partitions and emit tuples that match

Sort-Merge Join

- We already covered this (Lecture 2)

In case you forgot:

- You get to do this in your homework!

Sort-Merge Join

```
while not done {
  while (r < s) { advance r }
  while (r > s) { advance s }
  // assert r == s
  mark s // save start of "block"
  while (r == s) {
    // Outer loop over r
    while (r == s) {
      // Inner loop over s
      yield <r, s>
      advance s
    }
    reset s to mark
    advance r
  }
}
```

sid	sname
22	dustin
28	yuppy
31	lubber
31	lubber2
44	guppy
58	rusty

sid	bid
28	103
28	104
31	101
31	102
42	142
58	107

Sort-Merge Join

```
while not done {
  while (r < s) { advance r }
  while (r > s) { advance s }
  // assert r == s
  mark s // save start of "block"
  while (r == s) {
    // Outer loop over r
    while (r == s) {
      // Inner loop over s
      yield <r, s>
      advance s
    }
    reset s to mark
    advance r
  }
}
```

sid	sname
22	dustin
28	yuppy
31	lubber
31	lubber2
44	guppy
58	rusty

sid	bid
28	103
28	104
31	101
31	102
42	142
58	107

Sort-Merge Join

```
while not done {
  while (r < s) { advance r }
  while (r > s) { advance s }
  // assert r == s
  mark s // save start of "block"
  while (r == s) {
    // Outer loop over r
    while (r == s) {
      // Inner loop over s
      yield <r, s>
      advance s
    }
    reset s to mark
    advance r
  }
}
```

sid	sname
22	dustin
28	yuppy
31	lubber
31	lubber2
44	guppy
58	rusty

sid	bid
28	103
28	104
31	101
31	102
42	142
58	107

sid	sname	bid
28	yuppy	103

Sort-**Merge** Join

```
while not done {
  while (r < s) { advance r }
  while (r > s) { advance s }
  // assert r == s
  mark s // save start of "block"
  while (r == s) {
    // Outer loop over r
    while (r == s) {
      // Inner loop over s
      yield <r, s>
      advance s
    }
    reset s to mark
    advance r
  }
}
```

sid	sname
22	dustin
28	yuppy
31	lubber
31	lubber2
44	guppy
58	rusty

sid	bid
28	103
28	104
31	101
31	102
42	142
58	107

sid	sname	bid
28	yuppy	103
28	yuppy	104

Sort-**Merge** Join


```
while not done {
  while (r < s) { advance r }
  while (r > s) { advance s }
  // assert r == s
  mark s // save start of "block"
  while (r == s) {
    // Outer loop over r
    while (r == s) {
      // Inner loop over s
      yield <r, s>
      advance s
    }
    reset s to mark
    advance r
  }
}
```


sid	sname
22	dustin
28	yuppy
31	lubber
31	lubber2
44	guppy
58	rusty

sid	bid
28	103
28	104
31	101
31	102
42	142
58	107

sid	sname	bid
28	yuppy	103
28	yuppy	104

Sort-**Merge** Join


```
while not done {
  while (r < s) { advance r }
  while (r > s) { advance s }
  // assert r == s
  mark s // save start of "block"
  while (r == s) {
    // Outer loop over r
    while (r == s) {
      // Inner loop over s
      yield <r, s>
      advance s
    }
    reset s to mark
    advance r
  }
}
```


sid	sname
22	dustin
28	yuppy
31	lubber
31	lubber2
44	guppy
58	rusty

sid	bid
28	103
28	104
31	101
31	102
42	142
58	107

sid	sname	bid
28	yuppy	103
28	yuppy	104

Sort-**Merge** Join


```
while not done {
  while (r < s) { advance r }
  while (r > s) { advance s }
  // assert r == s
  mark s // save start of "block"
  while (r == s) {
    // Outer loop over r
    while (r == s) {
      // Inner loop over s
      yield <r, s>
      advance s
    }
    reset s to mark
    advance r
  }
}
```


sid	sname
22	dustin
28	yuppy
31	lubber
31	lubber2
44	guppy
58	rusty

sid	bid
28	103
28	104
31	101
31	102
42	142
58	107

sid	sname	bid
28	yuppy	103
28	yuppy	104
31	lubber	101

Sort-**Merge** Join


```
while not done {
  while (r < s) { advance r }
  while (r > s) { advance s }
  // assert r == s
  mark s // save start of "block"
  while (r == s) {
    // Outer loop over r
    while (r == s) {
      // Inner loop over s
      yield <r, s>
      advance s
    }
    reset s to mark
    advance r
  }
}
```


sid	sname
22	dustin
28	yuppy
31	lubber
31	lubber2
44	guppy
58	rusty

sid	bid
28	103
28	104
31	101
31	102
42	142
58	107

sid	sname	bid
28	yuppy	103
28	yuppy	104
31	lubber	101
31	lubber	102

Sort-**Merge** Join


```
while not done {
  while (r < s) { advance r }
  while (r > s) { advance s }
  // assert r == s
  mark s // save start of "block"
  while (r == s) {
    // Outer loop over r
    while (r == s) {
      // Inner loop over s
      yield <r, s>
      advance s
    }
    reset s to mark
    advance r
  }
}
```


sid	sname
22	dustin
28	yuppy
31	lubber
31	lubber2
44	guppy
58	rusty

sid	bid
28	103
28	104
31	101
31	102
42	142
58	107

sid	sname	bid
28	yuppy	103
28	yuppy	104
31	lubber	101
31	lubber	102
31	lubber2	101

Sort-**Merge** Join

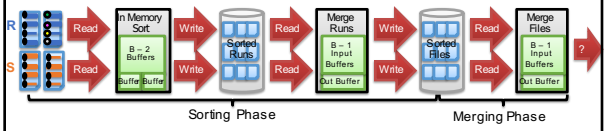
```
while not done {
  while (r < s) { advance r }
  while (r > s) { advance s }
  // assert r == s
  mark s // save start of "block"
  while (r == s) {
    // Outer loop over r
    while (r == s) {
      // Inner loop over s
      yield <r, s>
      advance s
    }
    reset s to mark
    advance r
  }
}
```

sid	sname
22	dustin
28	yuppy
31	lubber
31	lubber2
44	guppy
58	rusty

sid	bid
28	103
28	104
31	101
31	102
42	142
58	107

sid	sname	bid
28	yuppy	103
28	yuppy	104
31	lubber	101
31	lubber	102
31	lubber2	101
31	lubber2	102

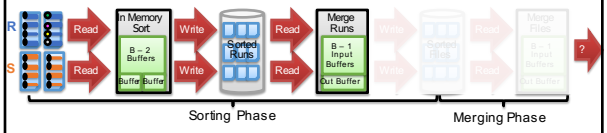
Cost of Sort-Merge Join



- Cost: Sort R + Sort S + ([R]+[S])
  - But in worst case, last term could be [R]*[S] (very unlikely!)
  - Q: what is worst case?
- Suppose buffer B > Sqrt(max([R], [S])):
  - Both R and S can be sorted in 2 passes
  - Total join cost = 4*1000 + 4*500 + (1000 + 500) = 7500
  - Comparisons:
    - Indexed Nested Loop: *80,500*
    - Block Nested Loop: *5,500*
    - Grace Hash Join: *4,500*

[R]=1000, p_R=100, |R| = 100,000
[S]=500, p_S=80, |S| = 40,000

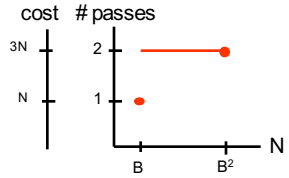
Sort-Merge Join Considerations



- An important refinement (needs 2x buffers):
  - Do the join during the final merging pass of sort
    1. Read R and write out sorted runs (pass 0)
    2. Read S and write out sorted runs (pass 0)
    3. Merge R-runs and S-runs, *while* finding R ⋈ S matches
  - Cost = 3*[R] + 3*[S]
- Sort-merge join an especially good choice if:
  - one or both inputs are already sorted on join attribute(s)
  - output is required to be sorted on join attributes(s)

Exploit excess memory

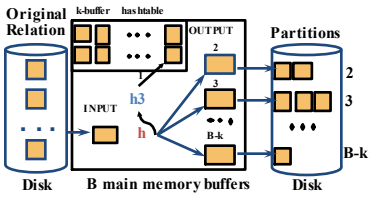
- Have B memory buffers
- Want to hash relation of size N blocks



If B < N < B², will have unused memory …

Hybrid Hashing

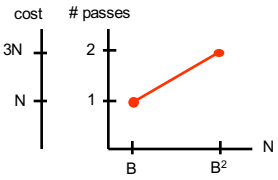
Idea: keep one of the hash buckets in memory!



Q: how do we choose the value of *k*?

Cost savings: hybrid hashing

- Now:



Hash Join vs. Sort-Merge Join

- Sorting pros:
  - Good if input already sorted, or need output sorted
  - Not sensitive to data skew or bad hash functions
- Hashing pros:
  - Can be cheaper due to hybrid hashing
  - For join: # passes depends on *size of smaller relation*
  - Good if input already hashed, or need output hashed

Recap

- Nested Loops Join
  - Works for arbitrary Θ
  - Make sure to utilize memory in blocks
- Index Nested Loops
  - For equi-joins
  - When you already have an index on one side
- Sort/Hash
  - For equi-joins
  - No index required
- No clear winners – may want to implement them all
- Be sure you know the cost model for each
  - You will need it for query optimization!