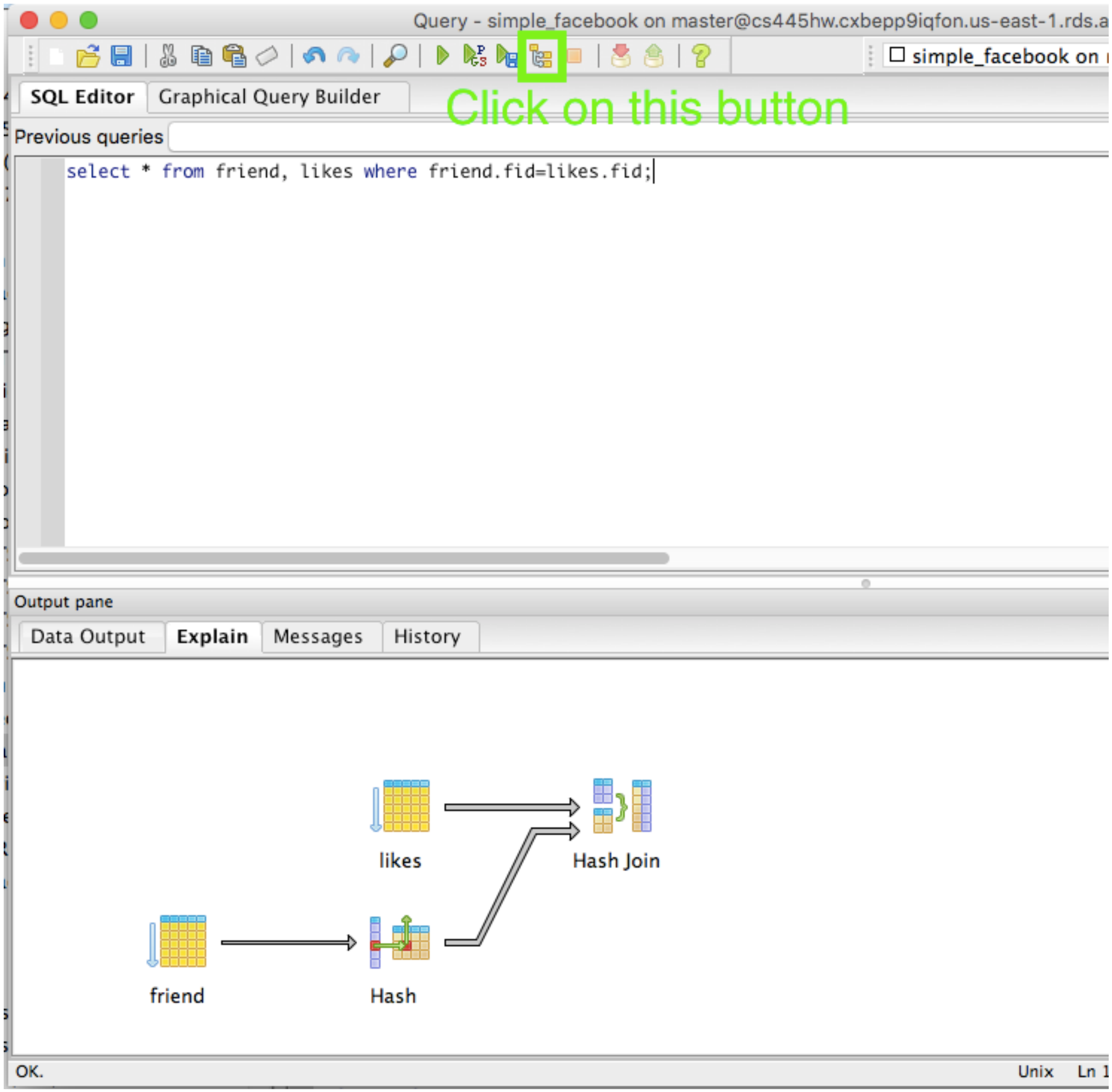
Outline

- ❖ Sorting
- ❖ Evaluation of joins
- ❖ Evaluation of other operations

EXPLAIN in PgAdmin3



```
----------------------- TPCH Q1: Single Relation ----------------------------
select
      l_returnflag,
      l_linestatus,
      sum(l_quantity) as sum_qty,
      sum(l_extendedprice) as sum_base_price,
      sum(l_extendedprice * (1 - l_discount)) as sum_disc_price,
      sum(l_extendedprice * (1 - l_discount) * (1 + l_tax)) as sum_charge,
      avg(l_quantity) as avg_qty,
      avg(l_extendedprice) as avg_price,
      avg(l_discount) as avg_disc,
      count(*) as count_order
from
      lineitem
where
      l_shipdate <= date '1998-12-01' - interval '82' day
group by
      l_returnflag,
      l_linestatus
order by
      l_returnflag,
      l_linestatus;
```

```
QUERY PLAN
----------
 Sort  (cost=4306256.71..4306256.73 rows=6 width=36)
   Sort Key: l_returnflag, l_linestatus
   ->  HashAggregate  (cost=4306256.53..4306256.63 rows=6 width=36)
         Group Key: l_returnflag, l_linestatus
         ->  Seq Scan on lineitem  (cost=0.00..1936078.65 rows=59254447 width=36)
               Filter: (l_shipdate <= '1998-09-10 00:00:00'::timestamp without time zone)
(6 rows)
```

```
-------------------- TPCH Q3: 3-way join-------------------------------------
select
	l_orderkey,
	sum(l_extendedprice * (1 - l_discount)) as revenue,
	o_orderdate,
	o_shippriority
from
	customer,
	orders,
	lineitem
where
	c_mktsegment = 'BUILDING'
	and c_custkey = o_custkey
	and l_orderkey = o_orderkey
	and o_orderdate < date '1995-03-22'
	and l_shipdate > date '1995-03-22'
group by
	l_orderkey,
	o_orderdate,
	o_shippriority
order by
	revenue desc,
	o_orderdate;
```

```
QUERY PLAN
----------
 Sort  (cost=4253633.68..4261644.36 rows=3204270 width=28)
   Sort Key: (sum((lineitem.l_extendedprice * (1::double precision - lineitem.l_discount)))),
orders.o_orderdate
   ->  GroupAggregate  (cost=3665934.82..3754052.25 rows=3204270 width=28)
         Group Key: lineitem.l_orderkey, orders.o_orderdate, orders.o_shippriority
         ->  Sort  (cost=3665934.82..3673945.50 rows=3204270 width=28)
               Sort Key: lineitem.l_orderkey, orders.o_orderdate, orders.o_shippriority
               ->  Hash Join  (cost=693200.74..3166353.39 rows=3204270 width=28)
                     Hash Cond: (lineitem.l_orderkey = orders.o_orderkey)
                     ->  Seq Scan on lineitem  (cost=0.00..1936078.65 rows=32176879 width=20)
                           Filter: (l_shipdate > '1995-03-22'::date)
                     ->  Hash  (cost=667234.93..667234.93 rows=1493745 width=12)
                           ->  Hash Join  (cost=60175.62..667234.93 rows=1493745 width=12)
                                 Hash Cond: (orders.o_custkey= customer.c_custkey)
                                 ->  Seq Scan on orders  (cost=0.00..455546.00 rows=7311526 width=16)
                                       Filter: (o_orderdate < '1995-03-22'::date)
                                 ->  Hash  (cost=55147.00..55147.00 rows=306450 width=4)
                                       ->  Seq Scan on customer  (cost=0.00..55147.00 rows=306450 width=4)
                                             Filter: (c_mktsegment = 'BUILDING'::bpchar)
(18 rows)
```

Some Common Techniques

- Algorithms for evaluating relational operators use some simple ideas extensively:
  - Indexing: Can use WHERE conditions to retrieve small set of tuples (selections, joins)
  - Iteration: Sometimes, faster to scan all tuples even if there is an index. (And sometimes, we can scan the data entries in an index instead of the table itself.)
  - Partitioning: By using sorting or hashing, we can partition the input tuples and replace an expensive operation by similar operations on smaller inputs.

* *Watch for these techniques as we discuss query evaluation!*

Schema for Examples

Sailors (*sid*: integer, *sname*: string, *rating*: integer, *age*: real)
Reserves (*sid*: integer, *bid*: integer, *day*: date, *rname*: string)

- **Reserves**:
  - Each tuple is 40 bytes long,
  - 100 tuples per page, (p_R)
  - 1000 pages. (M)
- **Sailors**:
  - Each tuple is 50 bytes long,
  - 80 tuples per page, (p_S)
  - 500 pages. (N)

Equality Joins With One Join Column

```
SELECT  *
FROM    Reserves R1, Sailors S1
WHERE   R1.sid=S1.sid
```

- In algebra: R $\bowtie$ S. Common relational operation!
  - R X S is large; R X S followed by a selection is inefficient.
  - Must be carefully optimized.

- We will consider more complex join conditions later.
- *Cost metric*: # of I/Os. We will ignore output costs.

Simple Nested Loops Join

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

- For each tuple in the *outer* relation R, we scan the entire *inner* relation S.
- Cost: $M + p_R * M * N$ = 1000 + 100*1000*500 = 1,000+ ($5 * 10^7$) I/Os.
- Assuming each I/O takes 10 ms, the join will take about 140 hours!

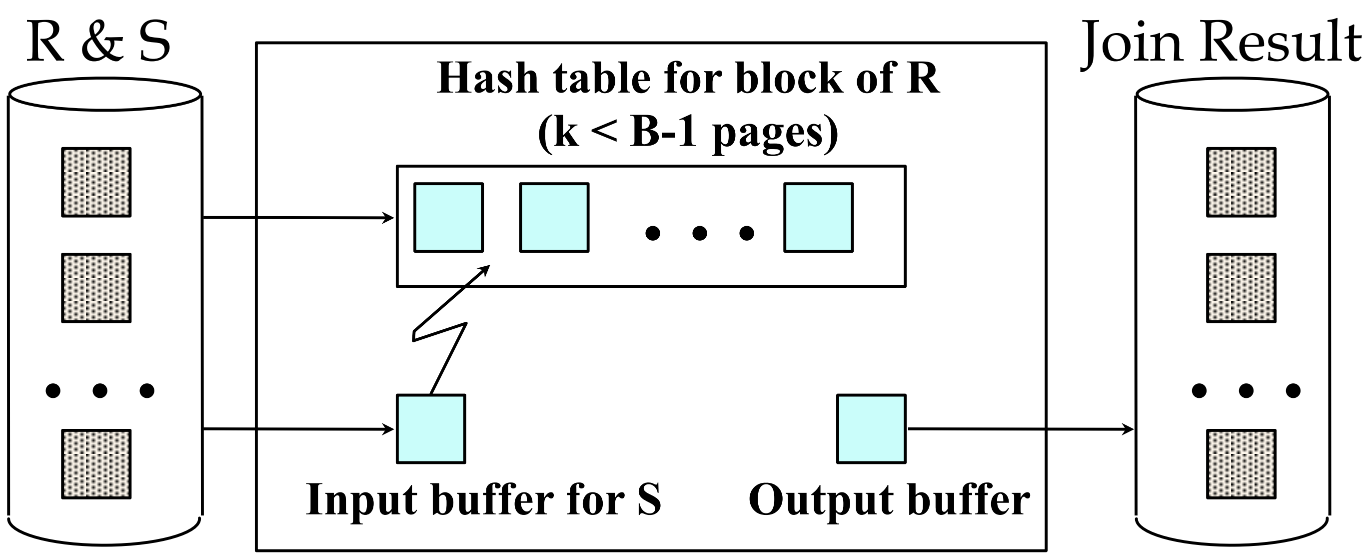
"Tuple at a time" Nested Loops Join

Page-Oriented Nested Loops Join

- For each *page* of R, get each *page* of S, and write out matching pairs of tuples <r, s>, where r is in R-page and S is in S-page.
  - Cost: M + M * N = 1000 + 1000*500 = 501,000 I/Os.
  - Assuming each I/O takes 10 ms, the join will take about 1.4 hours.
- Choice of the *smaller* relation as the *outer*
  - If smaller relation (S) is outer, cost = 500 + 500*1000 = 500,500 I/Os.

Block Nested Loops Join

- Take the smaller relation, say R, as outer, the other as inner.
- Use one buffer for scanning the inner S, one buffer for output, and use all remaining buffers to hold ``block'' of outer R.
  - For each matching tuple r in R-block, s in S-page, add <r, s> to result.



Examples of Block Nested Loops

- Cost: Scan of outer + #outer blocks * scan of inner
  - #outer blocks = $\lceil$ # pages of outer / block size $\rceil$
  - Given available buffer size B, block size is at most B-2.
  - M + N * $\lceil$ M / B-2 $\rceil$
- With Sailors (S) as outer, let block be 100 pages of S:
  - Cost of scanning S is 500 I/Os; a total of 5 *block*s.
  - Per block of S, we scan Reserves; 5*1000 I/Os.
  - Total = 500 + 5 * 1000 = 5,500 I/Os.
  - (a little over 1 minute)

Index Nested Loops Join

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

- If there is an index on the join column of one relation (say S), can make it the inner and exploit the index.
  - Cost: M + ((M*p_R) * cost of finding matching S tuples)
- For each R tuple, cost of probing S index is about 1.2 for hash index, 2-4 for B+ tree. Cost of then finding S tuples depends on clustering.
  - Clustered index: 1 I/O (typical).
  - Unclustered: up to 1 I/O per matching S tuple.

Examples of Index Nested Loops

- Hash-index on *sid* of Sailors (as inner):
  - Scan Reserves: 1000 page I/Os, 100*1000 tuples.
  - For each Reserves tuple: 1.2 I/Os to get data entry in index, plus 1 I/O to get (the exactly one) matching Sailors tuple.
  - Total: 1000+ 100*1000*2.2 = 221,000 I/Os.
- Hash-index on *sid* of Reserves (as inner):
  - Scan Sailors: 500 page I/Os, 80*500 tuples.
  - For each Sailors tuple: 1.2 I/Os to find index page with data entries, plus cost of retrieving matching Reserves tuples. If uniform distribution, 2.5 reservations per sailor (100,000 / 40,000). Cost of retrieving them is 1 or 2.5 I/Os (cluster?).
  - Total: 500+80*500*(2.2~3.7) = 88,500~148,500 I/Os.

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

- (1) Sort R and S on the join column, (2) Merge them (on join col.), and output result tuples.
- Merge: repeat until either R or S is finished
  - *Scanning*: Advance scan of R until current R-tuple>=current S tuple, advance scan of S until current S-tuple>=current R tuple; do this until current R tuple = current S tuple.
  - *Matching*: Now all R tuples with same value in Ri (*current R group*) and all S tuples with same value in Sj (*current S group*) match; output <r, s> for all pairs of such tuples.
- R is scanned once; each S group is scanned once per matching R tuple. (Multiple scans of an S group are likely to find needed pages in buffer.)

Example of Sort-Merge Join

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

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

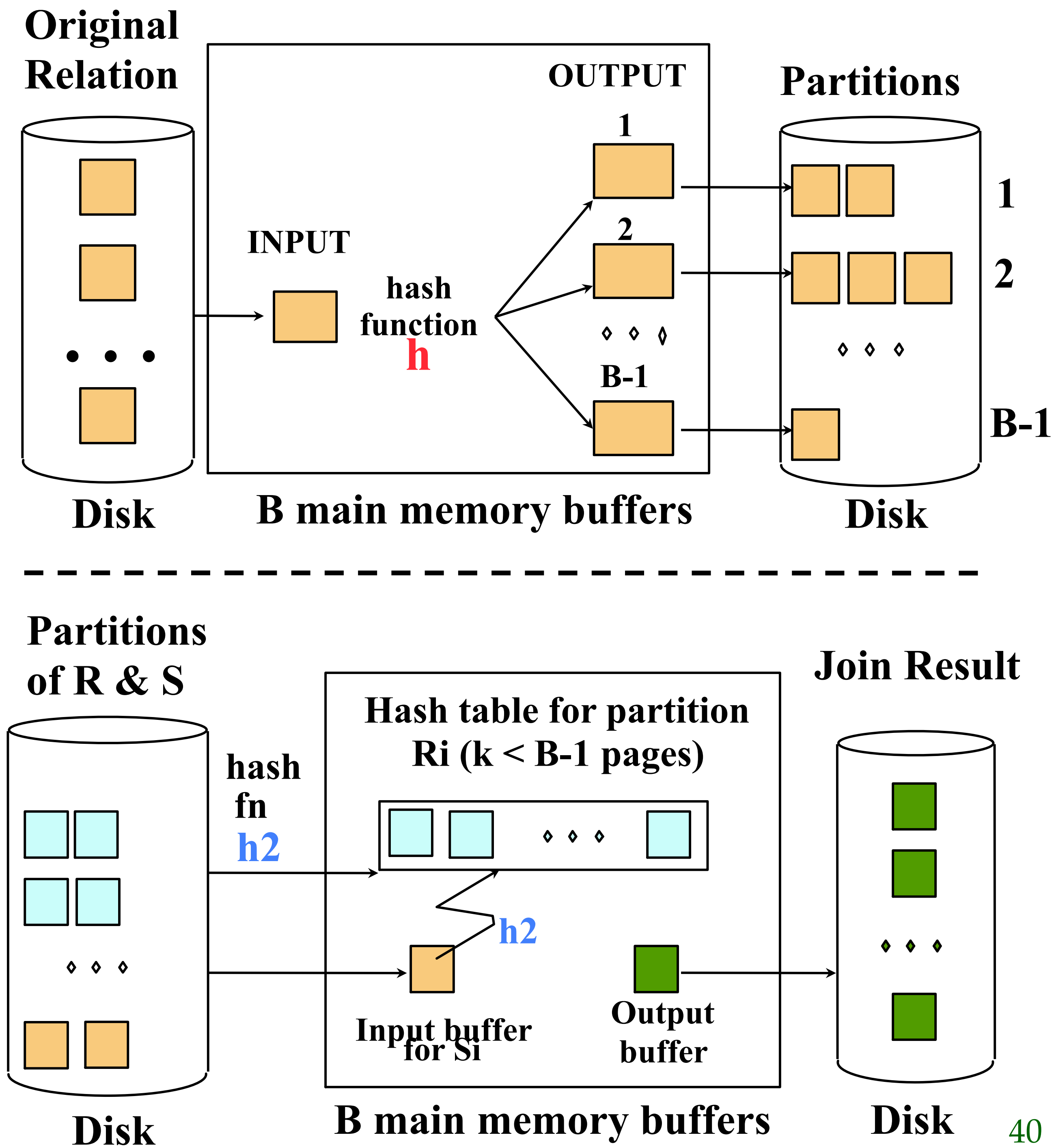
- Cost: $O(M\log_B M) + O(N\log_B N) + (M+N)$
  - The cost of merging, M+N, could be M*N (very unlikely!)
  - Notice that M+N is guaranteed in *foreign key join.*
  - As for external sorting, $\log_B M$ and $\log_B N$ are small numbers, e.g., 3, 4.
- With 35, 100 or 300 buffer pages, both Reserves and Sailors can be sorted in 2 passes; total join cost: 7500.

(BNL cost: 2500 (B=300), 5500 (B=100), 15000 (B=35))

Hash-Join

- Partitioning: Partition both relations using hash fn **h**: R tuples in partition i will only match S tuples in partition i.

- Probing: Read in partition i of R, build hash table on Ri using **h2 (<> h!)**. Scan partition i of S, search for matches.



Observations on Hash-Join

- # partitions ≤ B-1, and size of largest partition ≤ B-2 to be held in memory. Assuming uniformly sized partitions, we get:
  - M / (B-1) < (B-2), i.e., B must be > $\sqrt{M}$
  - Hash-join works if the smaller relation satisfies above.
- If we build an in-memory hash table to speed up the matching of tuples, a little more memory is needed.
- If hash function h does not partition uniformly, one or more R partitions may not fit in memory. Can apply hash-join technique recursively to do the join of this R-partition with corresponding S-

Cost of Hash-Join

- Partitioning reads+writes both relns; 2(M+N). Probing reads both relns; M+N I/Os. The total is 3(M+N).
  - In our running example, a total of 4500 I/Os using hash join, less than 1 min (compared to 140 hours w. NLJ).
- Sort-Merge Join vs. Hash Join:
  - With optimizations to Sort-Merge (not discussed in class), the cost is similar ~ 3(M+N)
  - Hash Join superior if relation sizes differ greatly.
  - Hash Join has been shown to be highly parallelizable.
  - Sort-Merge less sensitive to data skew; result is sorted.

General Join Conditions

- Equalities over several attributes (e.g., *R.sid*=*S.sid* AND *R.rname*=*S.sname*):
  - For Index NL, build index on <*sid, sname*> (if S is inner); or use existing indexes on *sid* or *sname* and check the other join condition on the fly.
  - For Sort-Merge and Hash Join, sort/partition on combination of the two join columns.
- Inequality conditions (e.g., *R.rname* < *S.sname*):
  - For Index NL, need B+ tree index.
    - Range probes on inner; # matches likely to be much higher than for equality joins (clustered index is much preferred).
  - Hash Join, Sort Merge Join not applicable.
  - Block NL quite likely to be a winner here.

Outline

- Sorting
- Evaluation of joins
- Evaluation of other operations