Evaluation of Relational Operations

CMPSCI 445

Spring 2018

Relational Operations

- We will consider how to implement:
  - *Selection* (σ) Selects a subset of rows from relation.
  - *Projection* (π) Deletes unwanted columns from relation.
  - *Join* ($\bowtie$) Allows us to combine two relations.
  - *Set-difference* ($-$) Tuples in reln. 1, but not in reln. 2.
  - *Union* ($\cup$) Tuples in reln. 1 and in reln. 2.
  - *Aggregation* (SUM, MIN, etc.) and GROUP BY
  - *Order By* Returns tuples in specified order.
- After we cover the operations, we will discuss how to *optimize* queries formed by composing them.

Outline

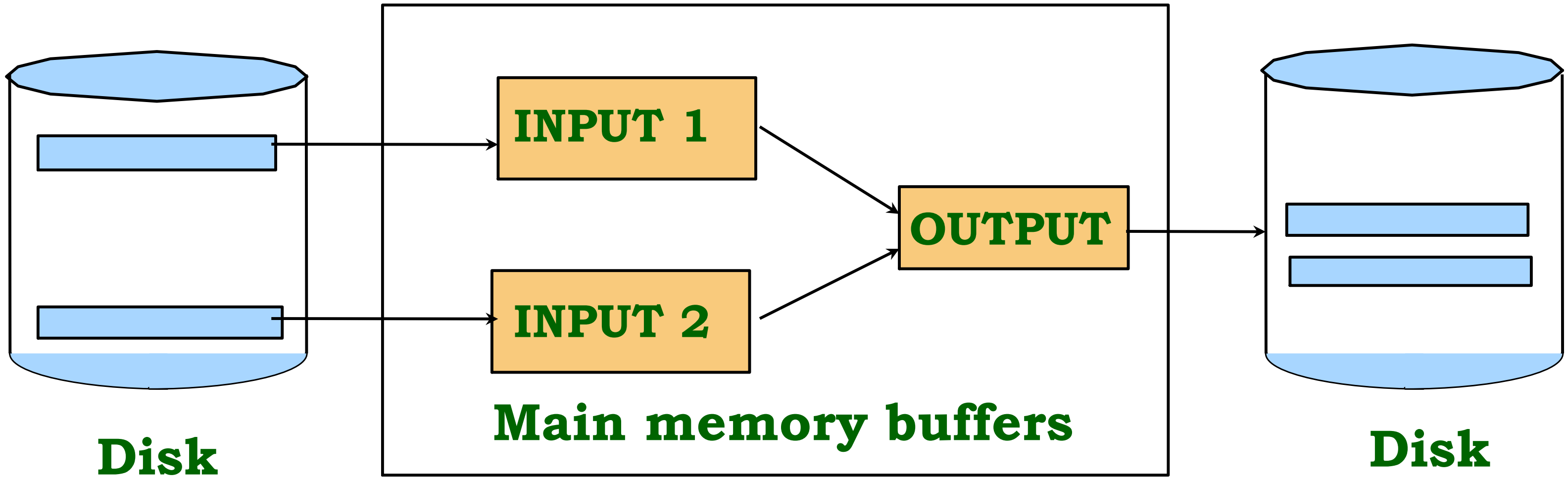
- Sorting
- Evaluation of joins
- Evaluation of other operations

Why Sort?

- A classic problem in computer science!
- Important utility in DBMS:
  - Data requested in sorted order (e.g., ORDER BY)
    - e.g., find students in increasing *gpa* order
  - Sorting useful for eliminating *duplicates* (e.g., SELECT DISTINCT)
  - *Sort-merge* join algorithm involves sorting.
  - Sorting is first step in *bulk loading* B+ tree index.
- Problem: sort 100Gb of data with 1Gb of RAM.

2-Way Sort: Requires 3 Buffers

- Pass 0: Read a page, sort it, write it.
  - only one buffer page is used
- Pass 1, 2, …, etc.:
  - three buffer pages used.



Two-Way External Merge Sort

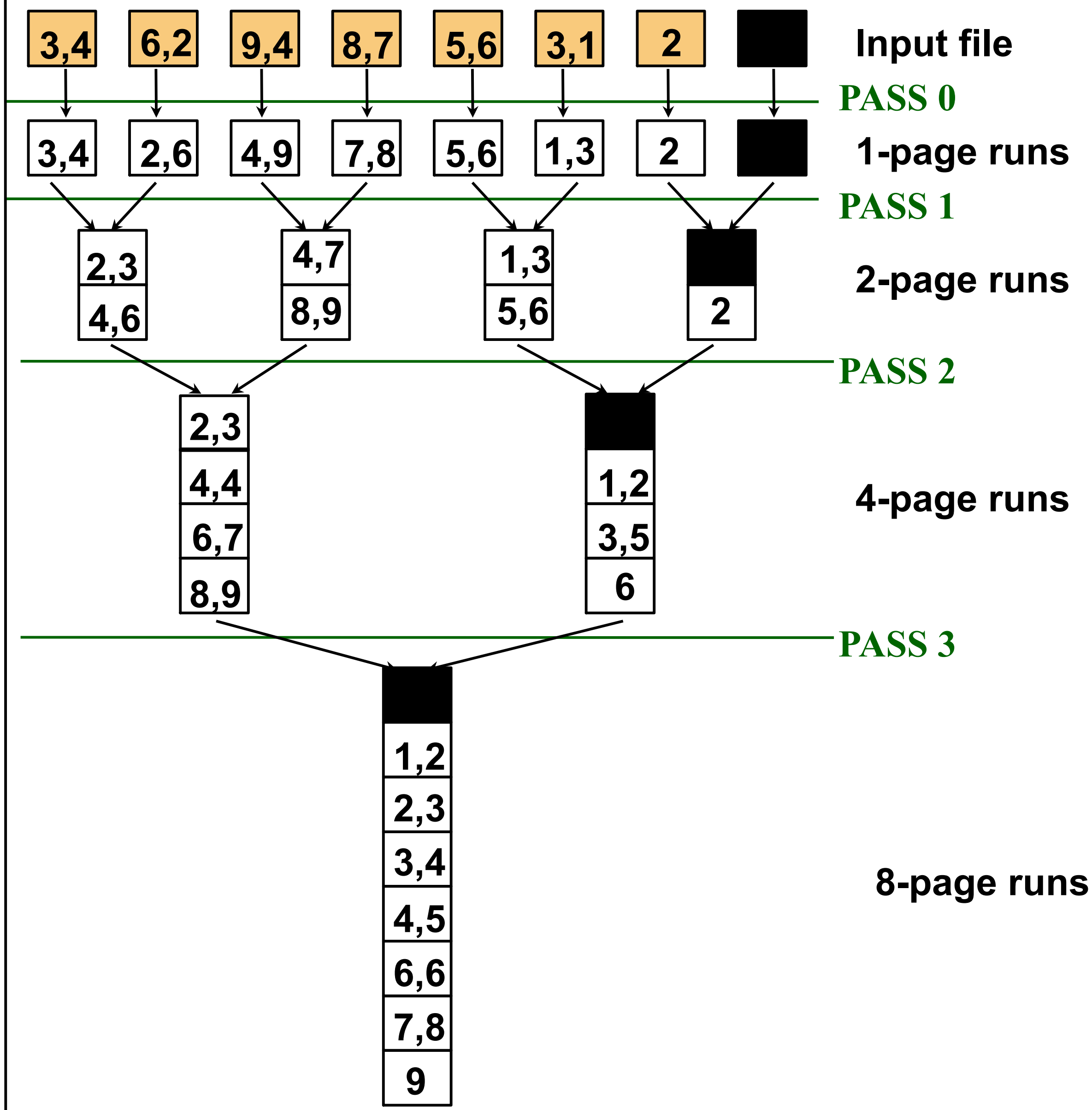
- Each pass we read + write each page in file: 2N.
- N pages in the file => the number of passes

  $= \lceil \log_2 N \rceil + 1$

- So total cost is:

$$2N\left(\lceil \log_2 N \rceil + 1\right)$$

- <u>*Idea:*</u> ***Divide and conquer:*** sort subfiles and merge

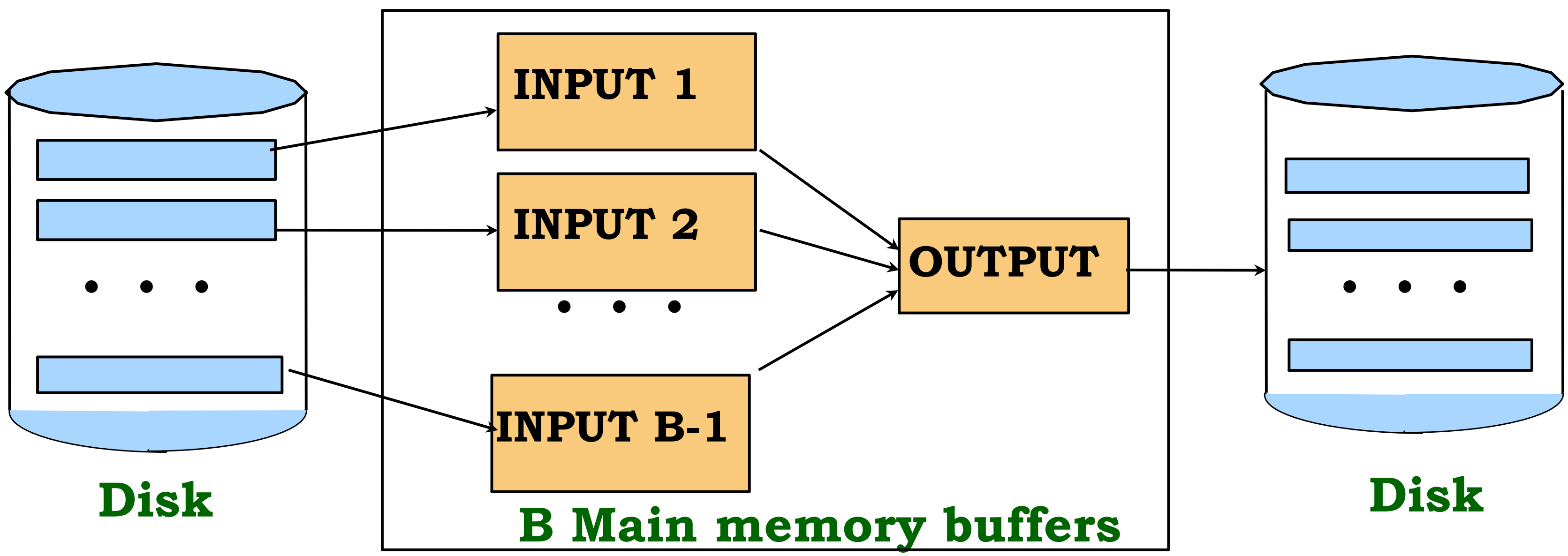


General External Merge Sort

☛ ***More than 3 buffer pages. How can we utilize them?***

❖ To sort a file with N pages using B buffer pages:

- Pass 0: use B buffer pages. Produce $\lceil N / B \rceil$ sorted runs of B pages each.
- Pass 2, …, etc.: merge B-1 runs.



iClicker 1

- For external merge sort using B buffers, the (B-1)-way merges produce runs of length (B-1).
  A. True
  B. False

Answer on next slide

iClicker 1

- For external merge sort using B buffers, the (B-1)-way merges produce runs of length (B-1).
  A. True
  B. False

Cost of External Merge Sort

- Number of passes: $1 + \lceil \log_{B-1} \lceil N / B \rceil \rceil$
- Cost = 2N * (# of passes)
- E.g., with 5 buffer pages, to sort 108 page file:
  - Pass 0: $\lceil 108 / 5 \rceil$ = 22 sorted runs of 5 pages each (last run is only 3 pages)
  - Pass 1: $\lceil 22 / 4 \rceil$ = 6 sorted runs of 20 pages each (last run is only 8 pages)
  - Pass 2: 2 sorted runs, 80 pages and 28 pages
  - Pass 3: Sorted file of 108 pages

Number of Passes of External Sort

N	B=3	B=5	B=9	B=17	B=129	B=257
100	7	4	3	2	1	1
1,000	10	5	4	3	2	2
10,000	13	7	5	4	2	2
100,000	17	9	6	5	3	3
1,000,000	20	10	7	5	3	3
10,000,000	23	12	8	6	4	3
100,000,000	26	14	9	7	4	4
1,000,000,000	30	15	10	8	5	4

Don’t forget: the full cost of external sort is 2N * (number of passes)

Sorting Records!

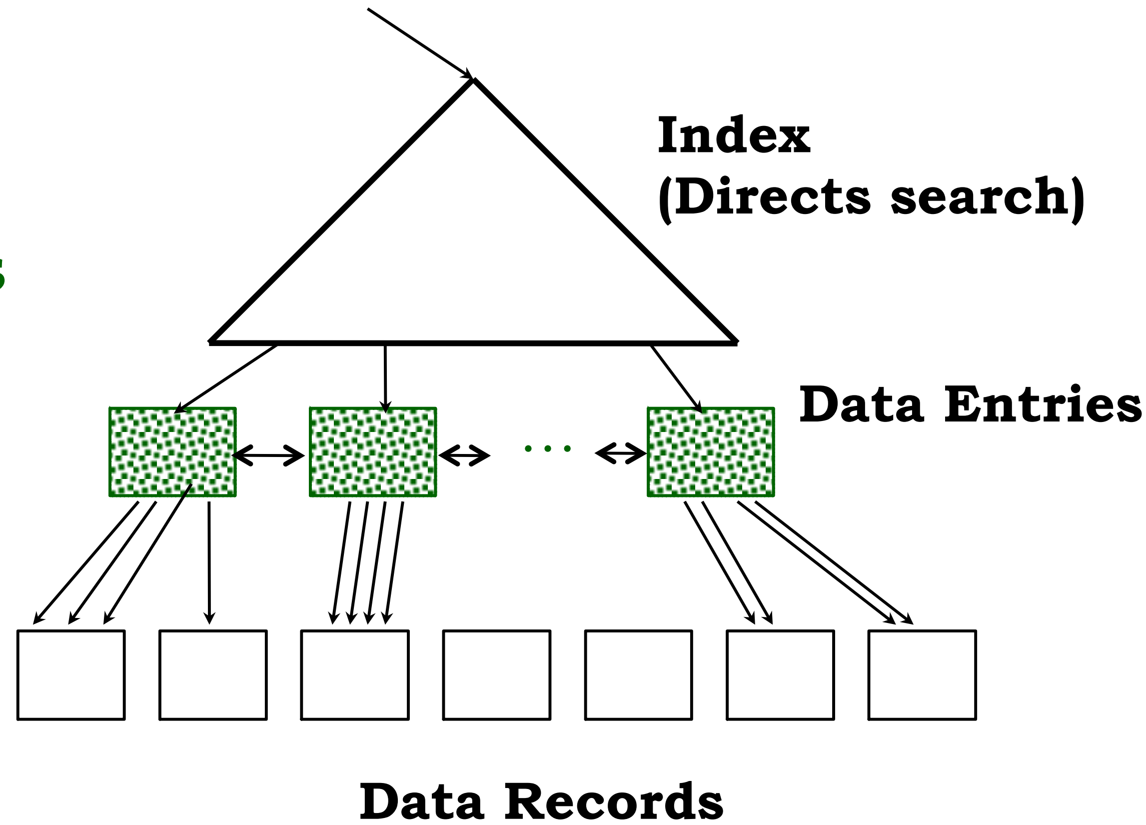
- Sorting has become highly competitive!
  - Parallel sorting is the name of the game ...
- Datamation sort benchmark: Sort 1M records of size 100 bytes
  - in 1985: 15 minutes
- New benchmarks proposed:
  - Minute Sort: How many can you sort in 1 minute?
  - Dollar Sort: How many can you sort for $1.00?

Using B+ Trees for Sorting

- Scenario: Table to be sorted has B+ tree index on sorting column(s).
- Idea: Can retrieve records in order by traversing leaf pages.
- ***Is this a good idea?***
- Cases to consider:
  - B+ tree is clustered — ***Good idea!***
  - B+ tree is not clustered — ***Could be a very bad idea!***

Clustered B+ Tree Used for Sorting

- Cost: root to the left-most leaf, then retrieve all leaf pages (if data entries are records)
- Otherwise, additional cost of retrieving data records: each page fetched just once.

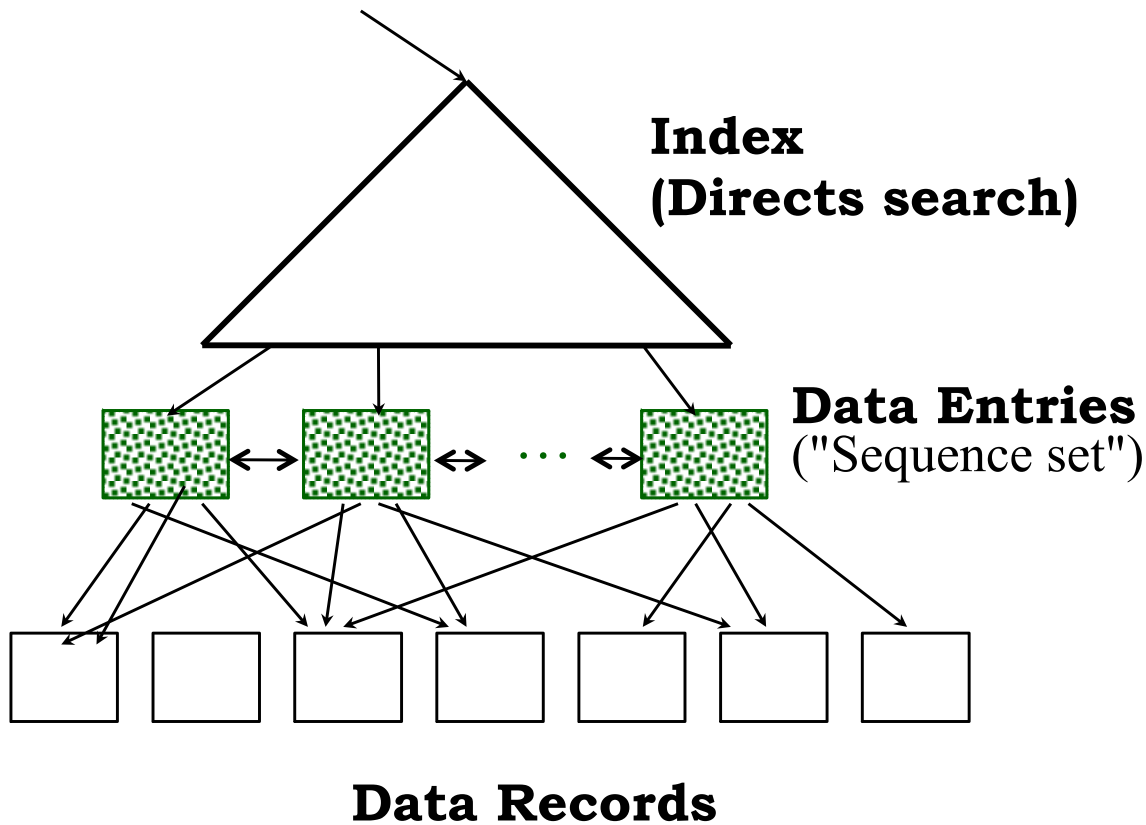


☛ *Always better than external sorting!*

Unclustered B+ Tree Used for Sorting

- Each data entry contains *rid* of a data record. In general, one I/O per data record!

Worse case I/O: pN
p: # records per page
N: # pages in file



Summary

- External sorting is important; DBMS may dedicate part of buffer pool for sorting!
- External merge sort minimizes disk I/O cost:
  - Pass 0: Produces sorted ***runs*** of size ***B*** (# buffer pages). Later passes: ***merge*** runs.
  - # of runs merged at a time depends on ***B***.
  - In practice, # of runs rarely more than 2 or 3.
- Clustered B+ tree is good for sorting; unclustered tree is usually very bad.

iClicker 2

- Consider scanning the N pages of a relation, searching for records satisfying some property (e.g. age = 21).
  - With 1 input buffer page and 1 output buffer page, this will require **N** IOs.
- With **B** buffer pages in total, how many IO's will it require?
  A. N
  B. N/B
  C. N/(B-1)
  D. $\log_B N$
  E. $\log_{B-1} N$

Answer on next slide

iClicker 2

- Consider scanning the N pages of a relation, searching for records satisfying some property (e.g. age = 21).
  - With 1 input buffer page and 1 output buffer page, this will require **N** IOs.
- With **B** buffer pages in total, how many IO's will it require?
  A. N
  B. N/B
  C. N/(B-1)
  D. $\log_B N$
  E. $\log_{B-1} N$

iClicker

- What's going to happen Wednesday
  A. Delayed opening (10 am)
  B. Campus will close early (12pm)
  C. Full all-day closure

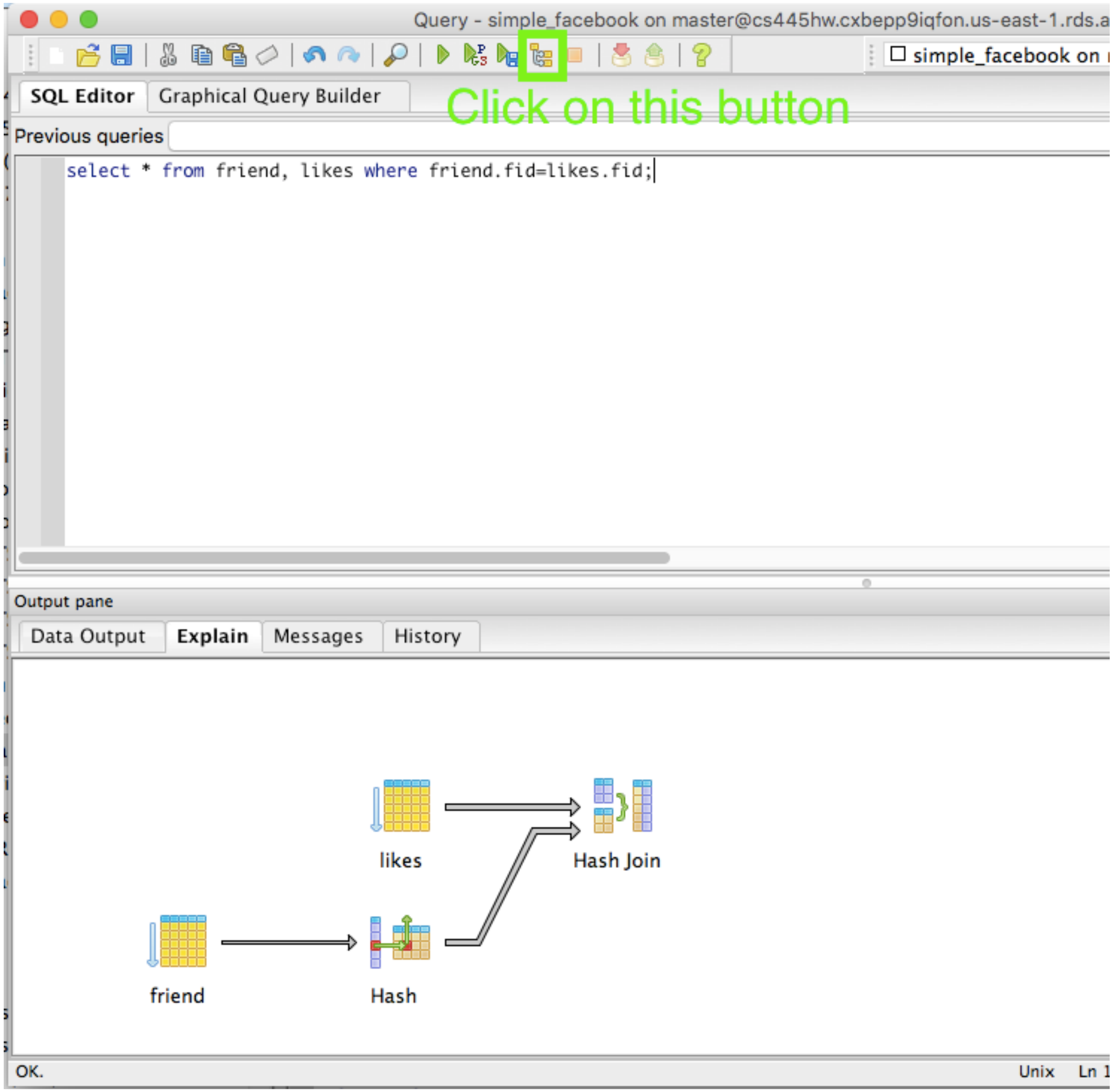
iClicker

- What’s going to happen Thursday
  A. Delayed opening (10 am)
  B. Other delayed opening (12pm?)
  C. Campus will close early (12pm)
  D. Full all-day closure

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

Equality Joins With One Join Column

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

- In algebra: R $\triangleright\triangleleft$ 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.