Joins

Holger Pirk

Housekeeping

Using the ODB query language

- You are very much encouraged to use it for the first task
  - it is C++ after all
- Not using it could result in a very slow program

Purpose of this lecture

You should

- Understand the different kinds of join semantics and when to apply which
- Understand the different nested subquery constructs and when to apply which
- Understand and know the basic join algorithms: nested loop (with variants), hash, sort-merge
- Be able to select an appropriate join implementation for a given situation

Joins

Why care?

Joins are everywhere

- In part due to to whole normalization business
  - These are mostly Foreign-Key joins (we'll talk about those in the context of indexing)
- In part because combining (joining) data produces value
  - These are more complicated (and interesting)

Examples

- Find users that have bought the same products
- Find the shortest route visting 5 of London's best sights
- Find advertisements that worked
  - (lead to users searching for a specific term within a timeframe)

Revision

We spoke about joins last week

- Joins are basically cross products with selections on top
  - `select R.r, S.s from R,S where R.id = S.id`

Revision

We spoke about joins last week

- Joins are basically cross products with selections on top
  - `select R.r, S.s from R,S where R.id = S.id`

Quick quiz: join or cross product

- `select R.r, S.s from R,S where R.r = R.id`

Revision

We spoke about joins last week

- Joins are basically cross products with selections on top
  - `select R.r, S.s from R,S where R.id = S.id`

Quick quiz: join or cross product

- `select R.r, S.s from R,S where R.r = R.id`

Quick quiz: join or cross product

- `select R.r from R,S where R.r = "something"`

Revision

We spoke about joins last week

- Joins are basically cross products with selections on top
  - `select R.r, S.s from R,S where R.id = S.id`

Quick quiz: join or cross product

- `select R.r, S.s from R,S where R.r = R.id`

Quick quiz: join or cross product

- `select R.r from R,S where R.r = "something"`

Quick quiz: join or cross product

- `select R.r from R,S where R.r = S.s`

Join formulation in SQL

Customer

CustomerID	Name	ShippingAddress
1	Holger	180 Queens Gate
2	Sam	32 Vassar Street
3	Peter	180 Queens Gate

Order

OrderID	CustomerID
1	1
2	2
3	3

Join formulations

Classic Join Syntax

```
select Name, OrderID from Customer, Order where Customer.CustomerID
     = Order.CustomerID
```

Explicit Join Syntax

```
select Name, OrderID from Customer JOIN Order on Customer.
    CustomerID = Order.CustomerID
```

"Natural" Join Syntax

```
select Name, OrderID from Customer NATURAL JOIN Order
```

Attribute-specific "Natural" Join Syntax

```
select Name, OrderID from Customer JOIN Order USING (CustomerID)
```

Inner joins

All of these have one thing in common

- They are Inner Joins

Inner Joins

- Every matching pair produces an output
- Not always what we want
- Introducing Outer Joins

Outer Joins

Why outer joins?

Customer

CustomerID	Name	ShippingAddress
1	Holger	180 Queens Gate
2	Sam	32 Vassar Street
3	Peter	180 Queens Gate
4	Daniel	180 Queens Gate

Order

ID	CustomerID
1	1
2	2
3	3
4	2

The question

- Who issued how many orders

A first shot at an SQL query

```
select name, count(*) from customer, [order]
where customer.id = [order].CustomerID group by name
```

Holger	1
Sam	2
Peter	1

Why outer joins?

A second shot

```
select name, count(*) from customer full outer join [order]
on (customer.CustomerID = [order].CustomerID) group by name
```

Holger	1
Sam	2
Peter	1
Daniel	1

- What went wrong here?

Why outer joins?

A third shot

```
select name, count([order].id) from customer full outer join [order
    ]
on (customer.CustomerID = [order].CustomerID) group by name
```

Holger	1
Sam	2
Peter	1
Daniel	0

- Success!!!

Left and Right Outer Joins

Left Join

A left join $R \overset{\text{L}}{\bowtie} S$ returns every row in R, even if no rows in S match. In such cases where no row in S matches a row from R, the columns of S are filled with NULL values.

Right Join

A right join $R \overset{\text{R}}{\bowtie} S$ returns every row in S, even if no rows in R match. In such cases where no row in R matches a row from S, the columns of R are filled with NULL values.

Outer Join

An outer join $R \overset{\text{O}}{\bowtie} S$ returns every row in R, even if no rows in S match, and also returns every row in S even if no row in R matches.

$$R \overset{\text{O}}{\bowtie} S \equiv (R \overset{\text{L}}{\bowtie} S) \cup (R \overset{\text{R}}{\bowtie} S)$$

Quiz

What are the bounds of for the size of the output (O) of this query

- `select * from R full outer join S on (R.r = S.s)`

Answers

A $|R| \leq |O| \leq |R| \times |S|$

B $max(|R|, |S|) \leq |O| \leq max(|R|, |S|, |R| \times |S|)$

C $|R| + |S| \leq |O| \leq max(|R| + |S|, |R| \times |S|)$

D $|S| \leq |O| \leq |R| \times |S|$

On matching criteria

The matching function

```
select * from R full outer join S on (R.r = S.s)
```

The matching function need not be equality

- If it is, we call the join an equi-join (these are the most important joins)
- If it is an inequality constraint ($<$ or $>$), we call them *inequality joins*
  - ```
    select count(*) from event, marker where
    event.time between marker.time and marker.time+60
    ```
- If it is an <>, we call it an *anti-join*
- All other joins are called *Theta joins*
    ```

Nested Subqueries

Nested Subqueries in from-clauses

Overview

- SQL allows queries in the from clause
- They behave exactly like base relations

```
select NH.name, count(*) from (select * from Customer where name <>
    "holger") as NH group by NH.name
```

Variable scoping rules

- Follow class programming language scoping
  - Nested queries have access to outer relations
- If the inner query uses variables of the outer, we call it correlated
  - Correlated subqueries can be nasty

Nested Subqueries in where-clauses

This breaks the set-based character of SQL

- we're combining a value of a tuple with a set

There are a couple of value to set comparison operators

- In
- Exists
- Some
- All

In

Definition

- true if the attribute value is contained in the set
- subquery must only return a single column

The simple case: statically defined sets

```
select * from Customer where name IN ("holger", "sam")
```

The correlated case

```
select Name from Customer where CustomerID IN (select CustomerID
    from Order)
```

equivalent to

```
select distinct Name from Customer, Order where Customer.CustomerID
     = Order.CustomerID
```

In

A more interesting case

- Find customers with more than five orders

```
select Name, OrderID from Customer where CustomerID IN
  (select CustomerID from Order group by CustomerID having count(*)
       > 5)
```

Exists

Definition

- true if the subquery returns a non-empty relation
- no restrictions regarding the schema of the subquery relation

Our join query strikes again

```
select CustomerID from Customer where exist (select * from Order
    where Order.CustomerID = Customer.CustomerID)
```

equivalent to

```
select distinct Name from Customer, Order where Customer.CustomerID
     = Order.CustomerID
```

Not exists for maximum calculation

- Find the customers with the longest names

```
select Name from Customer as cout where not exist (select * from
    Customer as cin where len(cin.name) > len(cout.name))
```

Some and All

Defintion

- Classic first order logic operators
  - Some is $\exists$
  - All is $\forall$

Example: Find customers that have orders before a threshold

```
select Name from Customer where customerID =some (select customerID
     from order where orderID < 3)
```

Example: Find customers that have all orders before a threshold

```
select Name from Customer where 3 >all (select orderID from order
    where order.CustomerID = Customer.customeID)
```

Technicalities

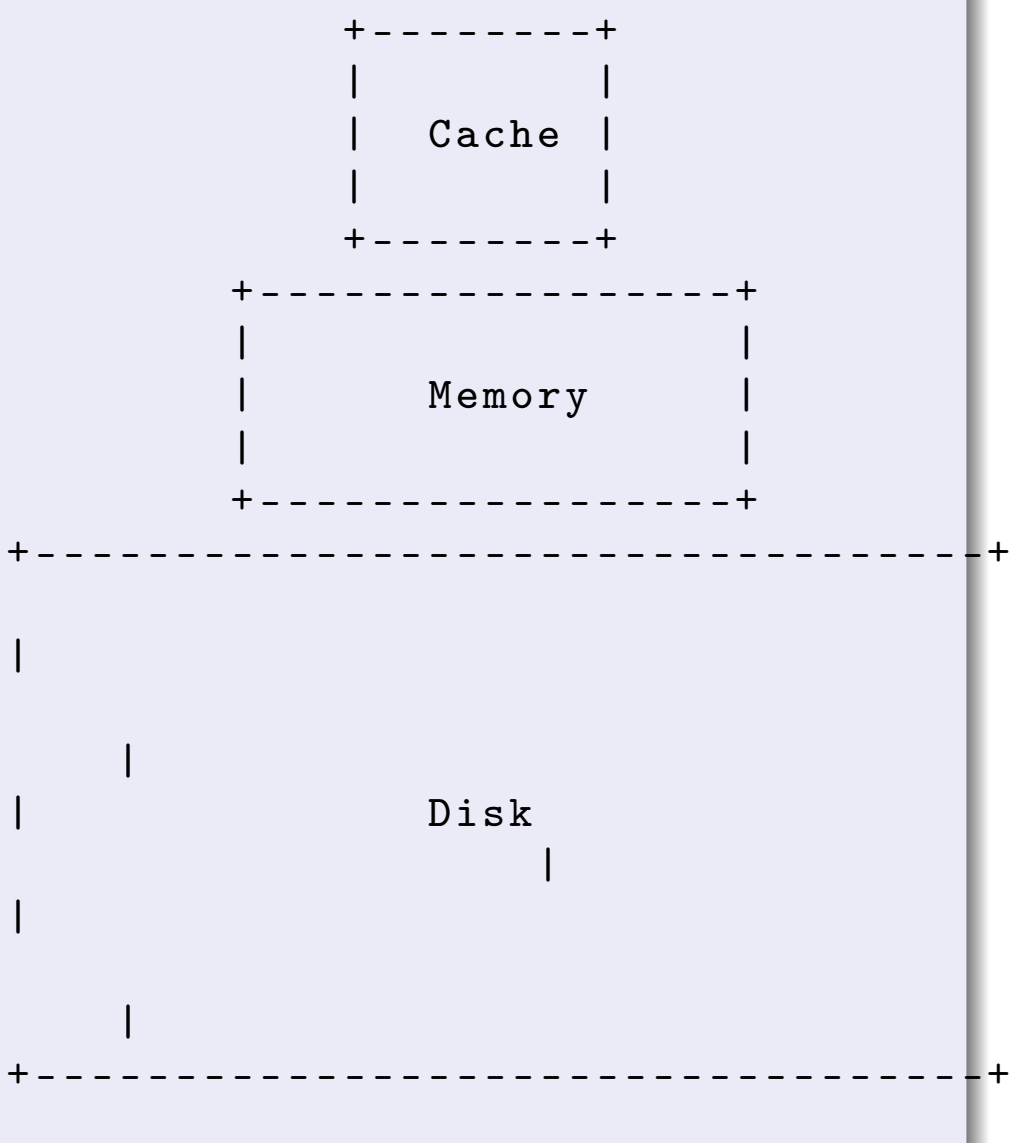
Nested subqueries are fun to write

- The are hard to evaluate for a DBMS (if they are correlated)
  - Conceptually, they need a re-evaluation of the subquery for every tuple in the outer query
- Queries can be decorrelated, i.e., rewritten into ones without correlated subqueries
  - Expressed in terms of joins
  - Decorellation is part of query optimization
  - There are efficient algorithms for evaluating joins

Join algorithms

Buffer Management

The memory hierarchy



Buffer Management

This is called Buffer Management in Databases

- The fast layer is called the *Buffer Pool*
  - This could be your cache or your memory
- Disk-based systems manage their own buffer pool
- Let's assume a simple interface:
  `readTupleFromPage("relationName", tupleID)`

Replacement strategy/Eviction policy

- The buffer pool has limited capacity (usually measured in pages)
  - If the buffer is full and a new page is brought in, which page has to go?
- A couple of alternatives
  - Least Recently Used (LRU)
  - Least Frequently Used (LFU)
  - Most Recently Used (MRU)
  - Random
  - ...

Nested Loop - the naive one

Nested Loop - the naive one

Implementation

```
for (size_t i = 0; i < leftRelationSize; i++) {
  auto leftInput = readTupleFromPage("left", i);
  for (size_t j = 0; j < rightRelationSize; j++) {
    auto rightInput = readTupleFromPage("right", j);
    if(leftInput[leftAttribute] == rightInput[rightAttribute])
      writeToOutput({leftInput, rightInput});
  }
 }
```

Example data

- R: 10, 17, 7, 16, 12, 8, 13
- S: 8, 16, 12, 1, 17, 2, 7

Nested Loop Join - the naive one

Properties

- Simple
- Sequential I/O
- Trivial to parallelize (no dependent loop iterations)

Effort

- $\Theta(|left| \times |right|)$
- Can be reduced to $\Theta(\frac{|left| \times |right|}{2})$ if value uniqueness can be assumed

Remember: Databases are I/O bound...

- ...well, they are on disk...
- ...so, lets see how many pages we need to read

Worksheet

Parameters

- Assume a nested loop join of two disk-resident tables in N-ary form
- The first table has 500,000 tuples of three integer attributes
- The second has 50,000 tuples of two integer attributes
- Pages are plain (unslotted) 4KB
- The page buffer can hold 10 pages
- The replacement strategy of the buffer manager is LRU

How many page replacements need to be performed

- 48,850,000
- 49 Million
- 97,850,000
- 25 Billion

Blocked Nested Loops - the slightly smarter

Blocked Nested Loops - the slightly smarter

Implementation (assuming all pages are filled)

```
auto tuplesOnLeftInputPages = 512;
auto tuplesOnRightInputPages = 1024;
for (size_t i = 0; i < leftRelationSize; i+= tuplesOnLeftInputPages
    )
  for (size_t j = 0; j < rightRelationSize; j+=
      tuplesOnRightInputPages)
    for (size_t i_page = 0; i_page < tuplesOnLeftInputPages; i_page
        ++) {
      auto leftInput = readTupleFromPage("left", i+i_page);
      for (size_t j_page = 0; j_page < tuplesOnRightInputPages;
          j_page++) {
        auto rightInput = readTupleFromPage("right", j+j_page);
        if(leftInput[leftAttribute] == rightInput[rightAttribute])
          writeToOutput({leftInput, rightInput});
      }
    }
```

Example data

- R: 10, 17, 7, 16, 12, 8, 13

Blocked Nested Loops - the slightly smarter

Properties

- Sequential I/O
- Trivial to parallelize (no dependent loop iterations)

Effort

- Same number of comparisons as nested loop
  - $\Theta(|left| \times |right|)$
  - Can be reduced to $\Theta(\frac{|left| \times |right|}{2})$ if value uniqueness can be assumed
- Better I/O behaviour

Worksheet

Parameters

- Assume a nested loop join of two disk-resident tables in N-ary form
- The first table has 500,000 tuples of three integer attributes
- The second has 50,000 tuples of two integer attributes
- Pages are plain (unslotted) 4KB
- The page buffer can hold 10 pages
- The replacement strategy of the buffer manager is LRU

How many page replacements need to be performed

- 95,740
- 95,746
- 97,845,924
- 97,850,000
- 97,850,142

Indexed Nested Loops - when you know what you're joining

Indexed Nested Loops - when you know what you're joining

Assumptions

- One side has an index
  - Let us, for now, assume it is sorted
- The other does not

Idea

- Scan the unindexed side (like you would in the nested loops case)
- Look up values on the inner side using the appropriate lookup function
  - In the case of a sorted relation, that means binary searching
  - We're going to talk about this more when we're talking about indexing

Indexed Nested Loops - when you know what you're joining

Properties

- Sequential I/O on the unindexed side
- Quasi-random on the indexed-side
- Parallelizable over the values on the unindexed side

Effort

- $\Theta(|unindexed| \times |\log indexed|)$
- $O(|left| \times |right|)$
- I/O behaviour on the indexed side is not-quite random
  - We'll talk about this when we talk about indexing

Sort-Merge Joins - a join for special cases

Sort-Merge Joins - a join for special cases

Implementation (assuming values are unique)

```
auto leftI = 0;
auto rightI = 0;
while (leftI < leftInputSize || rightI < rightInputSize) {
  auto leftInput = readTupleFromPage("left", leftI);
  auto rightInput = readTupleFromPage("right", rightI);
  if(leftInput[leftAttribue] < rightInput[rightAttribue])
    leftI++;
  else if(rightInput[rightAttribue] < leftInput[leftAttribue])
    rightI++;
  else {
    writeToOutput({leftInput, rightInput});
    rightI++;
    leftI++;
  }
 }
```

Example data

- R: 7, 8, 10, 12, 13, 16, 17
- S: ~1, 2, 7, 8, 12, 16, 17, ~

Sort-Merge Joins - a join for special cases

Effort

- $O(|left| \times \log |left| + |right| \times \log |right| + |left| + |right|)$
  - Assuming uniqueness

Properties

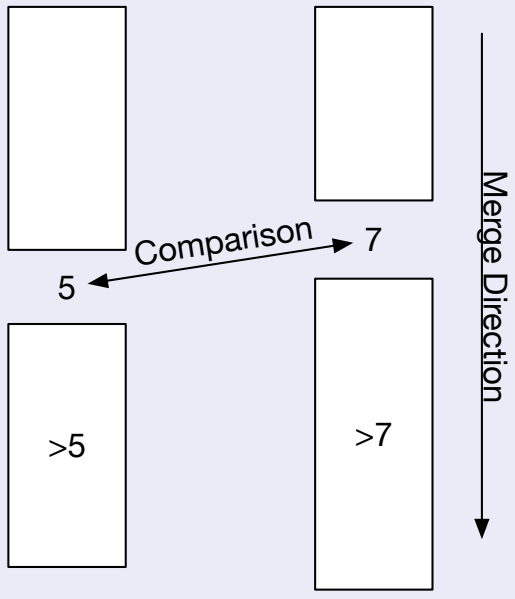
- Sequential I/O in the merge phase
- Tricky to parallelize
- Works for inequality joins
  - Careful when advancing the cursors

Why Sort-Merge Joins works

Invariants

- Assume, w.l.o.g., that the value on the right is less than the value on the left
- All values succeeding the value on the right are greater than the value on right
- $\Rightarrow$ No value beyond the value on the right can be a join partner
- $\Rightarrow$ The value on the left has no join partners succeeding the value on the right
- $\Rightarrow$ The cursor on the left can be advanced

Visualisation



Worksheet: Sort-Merge Joins

Parameters

- You have two sorted relations with a single attribute
  - R: `2, 5, 6, 7, 9, 11, 12`
  - S: `1, 4, 5, 6, 10, 11, 13`
- Evaluate the query
  `select r,s from R,S where s between r and r+2`
  - Assume an inclusive `between` (i.e., both bounds are included in the interval)
  - Use a sort-merge join
  - You can assume the values in each relation are unique

How many comparisons need to be performed

- 14
- 15
- 16
- 17
- 19

Hash joins - the special case that is so important

Hash joins - the special case that is so important

Implementation

```
extern int* hashTable;

for (size_t i = 0; i < buildSideSize; i++) {
  auto buildInput = readTupleFromPage("build", i);
  auto hashValue = hash(buildInput[buildAttribute]);
  while (hashTable[hashValue])
    hashValue = nextSlot(hashValue);
  hashTable[hashValue] = buildInput;
 }

for (size_t i = 0; i < probeSideSize; i++) {
  auto probeInput = readTupleFromPage("probe", i);
  auto hashValue = hash(probeInput[probeAttribute]);
  while (hashTable[hashValue] &&
         hashTable[hashValue][buildAttribute] != probeInput[
             probeAttribute])
    hashValue = nextSlot(hashValue);
  if(hashTable[hashValue][buildAttribute] == probeInput[
      probeAttribute])
    writeToOutput({hashTable[hashValue], probeInput});
 }
```

Hash join details... the hash function

Hash-function requirements

Pure no state

Known output domain we need to know the range of generated values

Contiguous output domain we do not want holes in the output domain

Nice to have

Uniform all values should be equally likely

Typical examples

- CRC32
- MD5
- MurmurHash
  - This is one of the fastest "decent" hash-functions
- Modulo-Division
  - Arguably the simplest function you could have

Conflict Handling

When a slot is already filled but there is space in the table...

- We need to put the value somewhere...
- The conflict handling strategy prescribes where

Requirements

- Locality (but not too much :-))
- No holes (probe all slots)

Many exist - let's talk about three

- Linear probing
- Quadratic probing
- Rehashing: specifically, cyclic group probing
- None of the above

Linear Probing

Description

- When a slot is filled, try the next one (distance 1)...
- ...and the next one (distance 2)...
- ...continue until you find one that is free (3,4,5,6, etc.)

Advantages

- Simple
- Good locality

Disadvantages

- Leads to long probe-chains for high-locality data
- For example, `9,8,7,6,5,4,3,2,2`

Hash-join with modulo hashing and linear probing

Implementation

```
extern int* hashTable;

for (size_t i = 0; i < buildSideSize; i++) {
  auto buildInput = readTupleFromPage("build", i);
  auto hashValue = buildInput[buildAttribute] % 10;
  while (hashTable[hashValue])
    hashValue = hashValue++;
  hashTable[hashValue] = buildInput;
 }

for (size_t i = 0; i < probeSideSize; i++) {
  auto probeInput = readTupleFromPage("probe", i);
  auto hashValue = probeInput[probeAttribute] % 10;
  while (hashTable[hashValue] &&
         hashTable[hashValue][buildAttribute] != probeInput[
             probeAttribute])
    hashValue = hashValue++;
  if(hashTable[hashValue][buildAttribute] == probeInput[
      probeAttribute])
    writeToOutput({hashTable[hashValue], probeInput});
 }
```

Quadratic Probing

Description

- When a slot is filled, try the next one (distance 1)...
- ...double the distance (distance 2)...
- ...continue until you find one that is free (4, 8, 16, etc.)

Advantages

- Simple
- Good locality in the first three probes
  - Bad after that (that is the point)

Disadvantages

- The first few probes are still likely to incur conflicts
- For example, `9,8,7,6,5,4,3,2,2`

Rehashing, specifically, cyclic group probing

The challenge:

- Randomize the probes
- Make them deterministic
- Make sure all slots are probed

Cyclic group probing

Hardcore math to the rescue

- There is a way to generate a (pseudorandom) sequence of numbers
  - while making sure every number in a range is generated
- A multiplicate group of integers modulo n
  - The equation is $f(x) = (x \times g) \mod n$ for certaing g and n
- How do we determine g and n (n is the size of our hash table)
  - $n = p^k$ with p an odd prime and $k > 0$
  - There is no easy way to calculate the *primitive roots* (g) for a given n
  - Using Mathematica: `PrimitiveRootList@11 = {2,6,7,8}`
  - $group = 6, 3, 7, 9, 10, 5, 8, 4, 2, 1, 6, ...$

Worksheet

Parameters

- You have two relations with a single attribute
  - Probe: `6, 7, 9, 12, 11, 15, 2`
  - Build: `1, 4, 5, 6, 15, 11, 14`
- Evaluate the query `select r,s from R,S where s = r`
  - Use a hash-join
  - You can assume the values in each relation are unique
- The hash function is $f(x) = x \mod 10 + 1$

Which is the best probing strategy in this case

- count the comparisons in the build phase
  - Linear Probing
  - Quadratic Probing
  - Rehashing with the cyclic group $f(x) = (x \times 6) \mod 11$

Hash joins - the special case that is so important

Properties

- Sequential I/O
- Parallelizable over the values on the probe side
- Parallelizing the build is tricky (Research opportunities!)

Effort

- $O(|build| \times |probe|)$ in the worst case
- $\Theta(|build| + |probe|)$ in the best case

Hash Joins practicalities

Cyclic groups are the theory...

- The practice is a lot less elegant
- People simply rehash using some arbitrary hash-function
  - (often the same they used for initial hashing)

Hashing is expensive

- Especially good hashing
  - Lots of CPU cycles (often more expensive than multiple data accesses)

Slots are often allocated in buckets

- Buckets are slots with space for more than one tuple
- You will sometimes see people implementing buckets as linked lists
  - A horrible idea if you care about lookup performance (inserts are okay)

Hash Joins practicalities

Hashtables are arrays too

- They occupy space
- They are usually overallocated by at least a factor two
- The are probed randomly in the probe phase (a lot)
  - You really want to make sure they stay in the buffer pool/cache
- For this class, assume that, if the hashtable does not fit, every access is a page fault
- Rule of thumb: use Hash Joins when one relation is much smaller than the other

How do you make sure of that?

- You partition!

Partitioning - when you don't have enough memory

Fundamental premise:

- Sequential access is much cheaper than random access
  - Difference grows with the page size
  - Random value access cost c
  - Sequential value access cost $\frac{c}{pagesize}$

Assume your hashtable does not fit in the buffer pool

- I.e., if the relation is larger than half the buffer pool
- It can pay off to invest in an extra pass for partitioning

Partitioning - an example

Assume

- Your smaller relation is 4 times the size of the buffer pool
- Partitioning it into 8 smaller partitions costs you one scan
  - Plus writing output
- Now, you join the large relation with each of the 8 smaller relations
  - Costs: 8 scans over the larger relation
- But: each of the hash lookups is now hitting the buffer pool
  - Disk seek latency: 3ms, RAM access latency: 30ns – a factor of 100K

Bonus

- You can parallelize the processing of each of the smaller joins
  - because they are disjoint
- You can partition the larger relation as well...
  - ...and only join the overlapping partitions

So, which join algorithm should I use?

My DBMS Professor:

- The answer is always sorting of hashing

The truth is more complicated

- Sort-Merge join if
  - Relations are sorted
  - Relations are unsorted but have similar size
  - If you are evaluating an inequality-join
- Indexed nested loop join if one relation has an index (e.g., is sorted)
- Hashjoin if one relation is much smaller than the other (less than 10%)
- Nested loop join if one relation is tiny (less than 20 values)
- Blocked nested loops join for theta-joins

The end