Implementing Joins

• A query on a single table can be implemented using either
  – table scan: scan the entire table for matching records
  – index scan: if the predicate includes a term for which an index exists

• What about joins?
Implementing Joins

• Suppose we change our students table and add a majors table as shown

```sql
CREATE TABLE students (  
sid BIGSERIAL PRIMARY KEY,  
name TEXT NOT NULL,  
year TEXT NOT NULL,  
mid INTEGER  
    REFERENCES majors(mid) 
);  

CREATE TABLE majors (  
mid SERIAL PRIMARY KEY  
title TEXT NOT NULL  
);
```

• And we execute a join query such as

```sql
SELECT * FROM students INNER JOIN majors USING (mid);
```
Nested Loop Join

• In a nested loop join
  – A scan is made of one table
  – For each record in the one table, a scan is made of the other table to find records that match on the join clause

• Runs in time $O(mn)$ where $m$ is the number of records in one table and $n$ is the number of records in the other
Block Nested Loop Join

• Since we are likely reading the records of the first table a block at a time, we can scan the records of the second table once per block instead of once per record.
Nested Loop Join

• When implementing an inner join or natural join we could further optimize by doing a single scan of the larger table and making the smaller table the one that is scanned repeatedly
  – Big gain if the smaller table can fit in memory

• When implementing an outer join then the "outer" table needs to be the outside one scanned
  – So we can include the records in the output that do not match any records in the "inner" table
Nested Loop Join

• We can potentially optimize further by filtering by the WHERE predicate first

```
SELECT * FROM students LEFT OUTER JOIN majors USING (mid)
WHERE students.year = '2010';
```

– If there is an index on students(year) then we can do an index scan to get the matching records from students and then only scan the majors table for those records

• And if there is another index on majors(mid) then we can use an index scan instead of a table scan on the majors table
Pipelining

• If we are first getting the students records using an index scan and then for each of those (or block of those) scanning the majors table (or index) then we have two choices
  – Choice 1: materialize the matching students records, either storing in memory (if small enough) or writing to a temporary file
  – Choice 2: as we get matching students records from the index, immediately scan the majors table for each students record (or block)
    • We never actually create the intermediate set of students records that match the WHERE predicate
    • This is called pipelining
Merge Join

• A *merge join* (also called a *sort-merge join*) is accomplished by
  – sorting the two relations by the join attributes
  – merging the two sorted relations

• For example, if we
  – sort the *students* records by *mid* and
  – sort the *majors* records by *mid* then
  – we can merge the two sorted relations in a single pass over both
Merge Join

• A merge join often requires *materializing* the sorted sets
  – Though if the join attributes match an index then that index can potentially be used to get the records in sorted order
  • For example, since the majors table has an index on mid we can use that to get the majors records in the order we need
    – However, we would still need to materialize the students records sorted by mid
Merge Join

• We can potentially reduce the size of the sets we need to materialize by filtering by the WHERE predicate first
  – For example, we only need to materialize and sort the students records where year = '2010'

• Pipelining is generally not an option going into a merge join as we need to materialize the sets before sorting
  – But we could potentially pipeline the output of the merge into the next stage
Hash Join

• To implement a *hash join*
  – Create a temporary hash table out of one relation (usually the smaller) using the join attributes as the key to the hash function
  – Scan the other table using the hash table to find matching records

• Similar to a nested loop join except we create a hash table of the "inner" table first

• Only works well if the hash table is small enough to keep in memory
Partition Hash Join

• A partition hash join can be used with larger tables
  – For each of the two tables being joined
    • Create a hash table using the join attributes as the key to the hash function
    • Partition the hash table using the hash function, storing the different partitions in different temporary files
  – For each partition, join with the matching partition from the other hash table