# Merging Sorted Lists

- Example!

# Group-By Aggregation

## What if you want multiple aggregate values?

- **SELECT A, SUM(B) FROM R**
  - Creates one row for each A, with a sum of all of the B values from rows with that A.
- **How do we implement this?**

## Idea 1: In-Memory Hash Table

- **Scan records in any order**
- **For each record, check to see if the hash table contains the group by attribute(s) value(s)**
  - If not, create a new entry in the hash table with the default group value
- **Incorporate the new record's aggregate value**

## Idea 2: Pre-Sort the Data

- **Problem w/ Idea 1: What if you run out of memory**
- **Use the external sort algorithm above by the group-by attributes**
- **Benefit: you know that all elements of a single group will be adjacent to one another:**
  - If you iterate over the sorted list of elements, as soon as the group by attributes change, you know you're done with that group
  - ... so you only ever need to keep one "current value" in memory at a time
- **Pro: You can start emitting intermediate results before you're done with everything**
- **Con: Log(N) full passes over the data**

## Idea 3: Pre-Hash the Data

- **Do one pass through the data to create hash buckets that will fit in memory**
- **Like sorting, but you only need one pass through the data**
  - ... unless you guess wrong about the number of buckets to create

# Joins and Cross Products

## How do you combine 2 tables?

- **Merge rows (A U B)**
- **Merge columns**
  - **Question**: What rows from A go with what rows from B?
  - Example
    - Data
      - Table of Students(student_id, name)
      - Table of Courses(course_id, title)
      - Table of SignedUpFor(student_id, course_id)
    - Count the number of students signed up for each course?
      - SELECT title, COUNT(*) FROM Courses NATURAL JOIN SignedUpFor
    - Count the number of people named "Kirk" signed up for each course?
      - SELECT title, COUNT(*) FROM Courses NATURAL JOIN SignedUpFor NATURAL JOIN Students WHERE name LIKE '% Kirk'
- **General Pattern**
  - Pair rows from A with rows from B where a specific condition holds (e.g., Courses.course_id = SignedUpFor.course_id)
  - More general conditions are also possible
    - "List identification numbers of borrowers who took out books on two different days"
      - Join Borrower with itself on "borrower.1id = borrower2.id AND borrower1.date <> borrower2.date"
    - "Find all restaurants within 2 miles of each person"
      - WHERE distance(person.loc, restaurant.loc) < 2 miles

## How do you implement this?

- **(Naive) Idea 1: Nested Loop Join**
  - Try every pair of tuples against the condition
    - foreach(tuple1 in left)

- foreach(tuple2 in right)
  - if(condition(tuple1, tuple2))
    - emit(concat(tuple1 + tuple2))
- Slow... but guaranteed to work on any condition
  - O(N^2)

- **(Slighlty less naive) Idea 2: Block Nested Loop Join**
  - Limitation of Idea 1: Inner loop loads ALL of the data in |left| times
  - Idea: Load in Blocks
  - foreach(block1 in left)
    - foreach(block2 in right)
      - foreach(tuple1 in block1)
        - foreach(tuple2 in block2)
          - if(condition(tuple1, tuple2))
            - emit(concat(tuple1 + tuple2))
  - Slightly faster... only need to load in |left| / |block| copies
    - Still O(N^2), but with a better constant
- **Idea 3: Sort + Merge (Sort-Merge Join)**
  - If you have a predicate of the form A = B
    - Sort left on A, sort right on B, and then merging is linear
  - foreach(tuple in merge(condition, sort(left), sort(right))):
    - if(condition(tuple1, tuple2))
      - emit(concat(tuple1 + tuple2))
  - Total cost: Cost of sorting + O(N)
    - Data might already be sorted!
    - Otherwise, O(N*log(N))
  - Limitation: Only works if you have an A = B predicate (so you can sort on A, B)
- **Idea 4: Use an Index (Index-Nested Loop Join)**
  - foreach(tuple1 in left)
    - foreach(tuple2 in right.index_lookup(condition, tuple1))
      - if(condition(tuple1, tuple2))
        - emit(concat(tuple1 + tuple2))
  - |left| index lookups rather than full table scans
    - O(N * [cost of one index lookup])
- **Idea 5: Build an Index... in memory (1-pass index join)**
  - left_index = {}
  - foreach(tuple1 in left)
    - left_index.add(tuple1)
  - foreach(tuple2 in right)
    - foreach(tuple1 in left_index.index_lookup(condition, tuple2))
      - if(condition(tuple1, tuple2))
        - emit(concat(tuple1 + tuple2))
  - Works with Tree indexes, Hash indexes
  - Overall Cost: O(N logN) or O(N)
    - Cost of building index (O(N logN) for tree, O(N) for hash
    - Cost of scanning, per-record: O(logN) for tree, O(1) for hash
    - Might need to return multiple records... so really it's O(logN + |records returned|) and O(1+ |records returned|)
  - Most efficient algorithm available... but requires enough memory for at least one table to stay in memory
- **Idea 6: Build an index on disk (2-pass index join)**
  - Same as before, but index goes to disk
  - **Problem**: Random access to disk can be avoided!
    - Solution: Build an index on **both** inputs
  - For a hash index, make sure you use the same hash fn for both tables.
  - For a tree index... welll... this basically degenerates to Sort+Merge Join
  - Cost: O(N) IOs for Hash ... but with a fairly high constant (join adds 2 IOs per input page)