Recap

- Data-savviness is the future!
- Notion of a DBMS
- The relational data model and algebra: bags and sets
- SQL Queries
- SQL Modifications
- SQL DDL
- Database Design
- Views, constraints and triggers
- Indexes
- Next: query processing

Given a SQL query Q

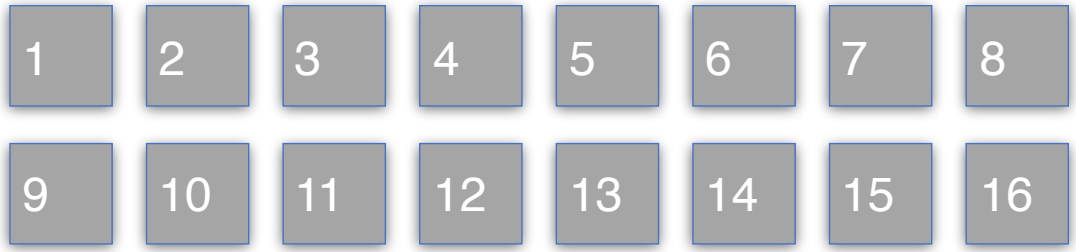
- How should Q be executed by the DBMS?
- What do we (the system) know?
  - We know Q, therefore we know the relations it is operating on, the predicates, the grouping and aggregating attributes
  - We know indexes on the relations, the sizes of the relations, statistics about the relations (# of columns and distinct values)…
- Goal: come up with a query execution plan:
  - We think of plans at two levels
    - The logical level: at the level of operators $\sigma, \Pi, \bowtie$
    - **The physical level: specific implementations**

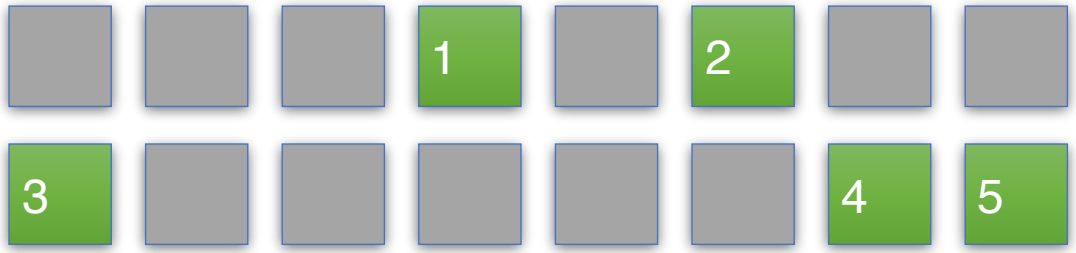
Logical Operators vs. Physical Operators

- Logical operators are relational algebra (RA) operators
  - Describe "what" is done
  - e.g., union, select, grouping, project
    - We covered only relational algebra operators for the basic operators, but there are operators for grouping and sorting as well
- Physical operators describe implementations of these operators
  - Describe "how" to do it
  - e.g., for join
    - nested-loop, sort-merge, hash join, …
  - Physical operators also pertain to non-RA operators such as scanning a table

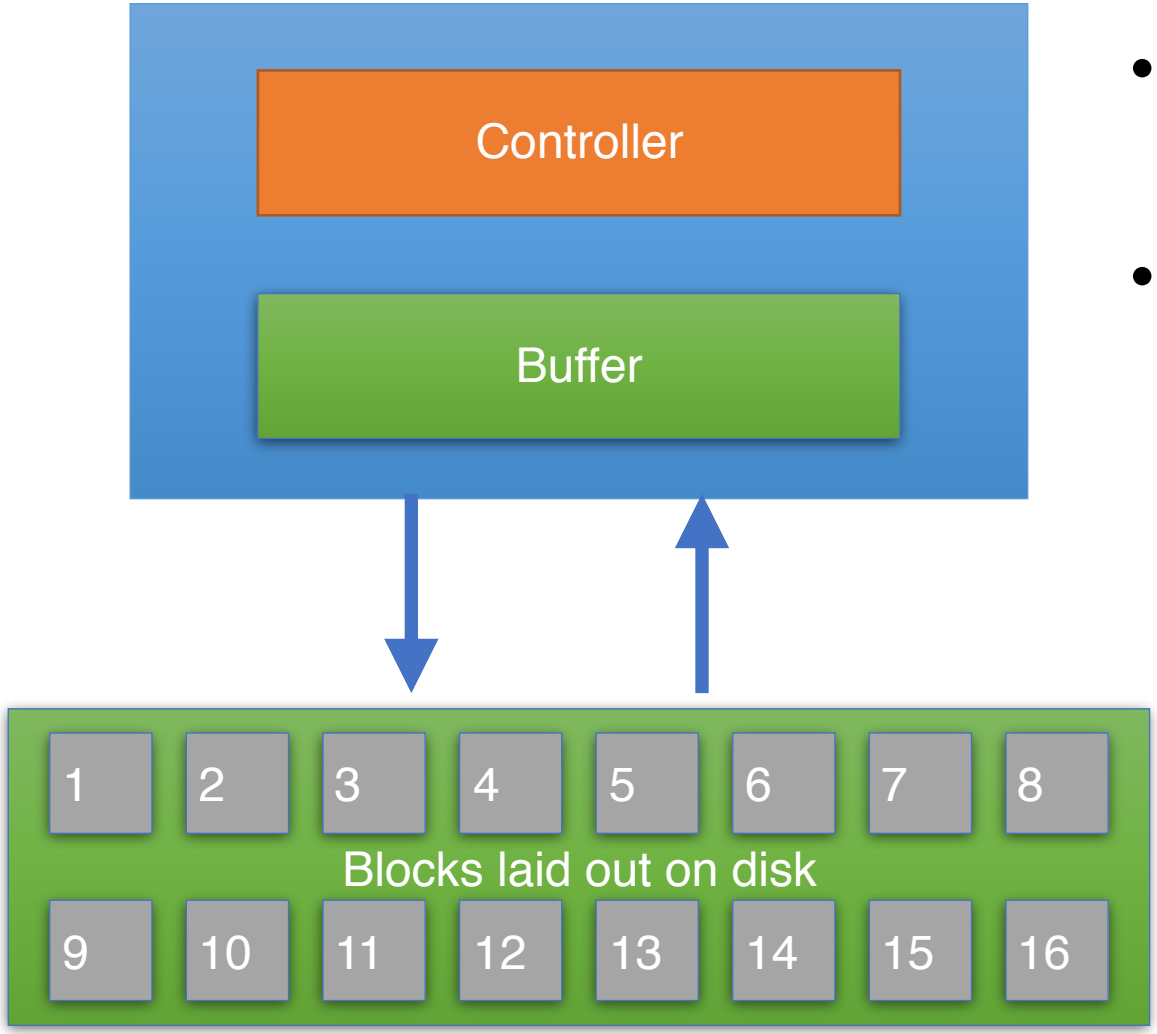
Getting started: Scanning a Relation R

- Either all of R or parts of it that satisfy some condition
- Simple Table Scan: Read all blocks on disk that contain tuples of R one-by-one
- Index Scan: Read the index, and use it to read relevant blocks of R
  - More valuable if only a small fraction of R is "relevant"



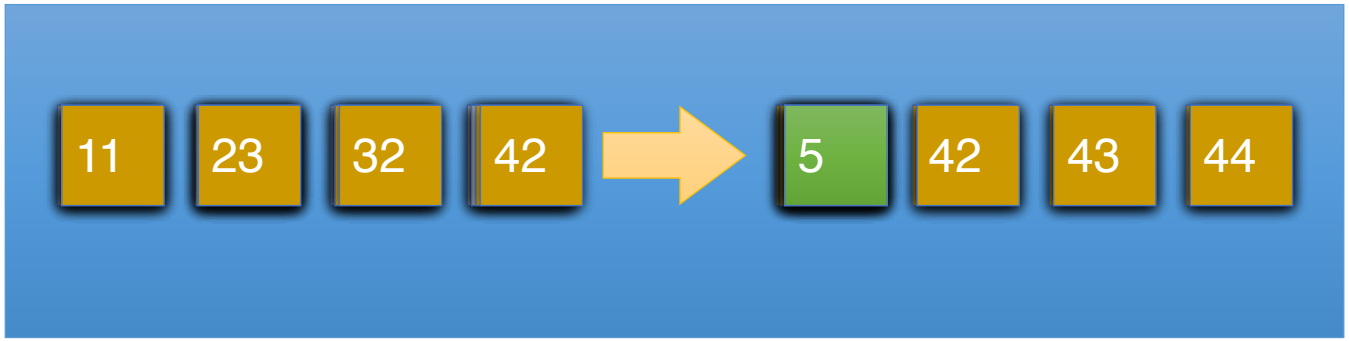


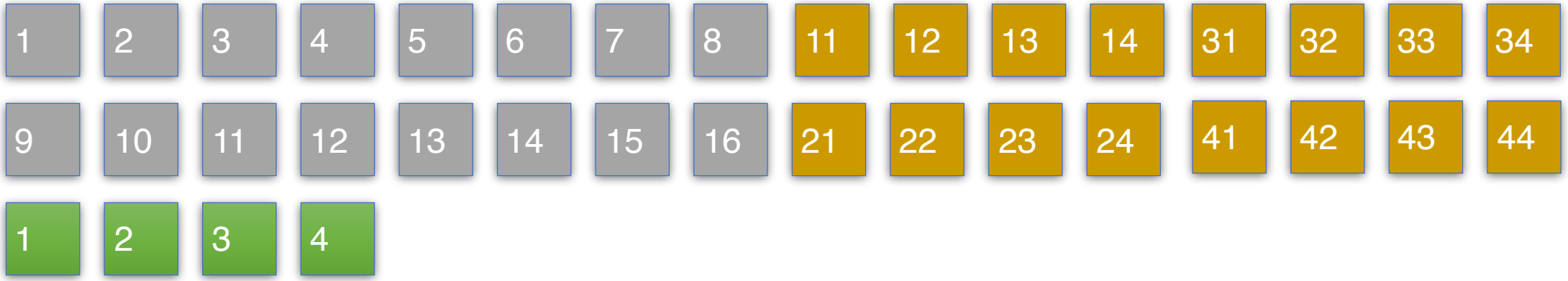
Getting Started 2: Sorting a Relation R



- Case 1: R fits in the buffer
  - Easy, sort in memory
- Case 2: R does not fit in the buffer
  - Q: How would you go about sorting?
  - A: 2-pass (or multi-pass) merge sort

Two-Pass Merge Sort Intuition





Two-Pass Merge Sort

- Say B(R) is the number of blocks of R
- M is the number of blocks in the buffer
- Pass 1:
  - Sort B(R)/M subsets of M blocks of R each
  - Each is output to disk as a *run*
- Pass 2:
  - "Merge" these runs by bringing in one block for each of them

- Variant
  - For Pass 2: If you can't fit one block for each run in the buffer, repeatedly do this: multi-pass merge sort

Example

M = 3, each block holds two values

Data on disk: [3,9] [8,1] [7,6] [9,5] [2,0] [3,8]

Step 1: Sort Runs

[3,9] [8,1] [7,6] => [1,3] [6,7] [8,9]

[9,5] [2,0] [3,8] => [0,2] [3,5] [8,9]

Step 2: Merge Sorted Runs

[1,3][0,2][_,_] => [1,3][**0**,2][0,_] => [**1**,3][**0**,2][0,1] => output [0,1]

[**1**,3][**0**,2][_,_] => [**1**,3][**0,2**][2,_] => [**1**,3][3,5][2,_] => [**1**,**3**][3,5][2,3] => output [2, 3]

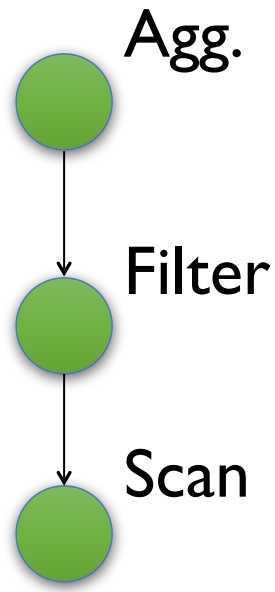
[6,7][3,5][_,_] => [6,7][**3**,5][3,_] => [6,7][**3**,**5**][3,5] => output [3,5]

[6,7][**3**,**5**][_,_] => [6,7][8,9][_,_] => [**6**,7][8,9][6,_] => [**6**,**7**][8,9][6,7] => output [6,7]

[**6**,**7**][8,9][_,_] => [8,9][8,9][_,_] => [**8**,9][**8**,9][8,8] => output [8,8]

Now, other operators beyond scan & sort …

- A brief aside first.
- Physical operators are implemented following the iterator model
- Each operator operates (i.e., iterates on tuples) independent of others, requests tuples from operators below
- So no operator waits for the operator below to finish execution before continuing = *pipelining*
- Exceptions are "blocking" operators
  - Aggregation, sorting, …



Let's Talk About Joins

- Joins (theta or natural) are expensive
  - At a high level, need to correlate information across multiple relations
- Many many different ways to do joins
- We'll talk about 3 different ways…. to show you how complex it is!

- We won't talk about how to pick between these ways… that is the job of the DBMS to work out a "cost" estimate for each way and pick one with lowest cost

- Instead, goal is to give you a vocabulary and intuition for various algorithms

Join Approach 1: Nested Loop Joins

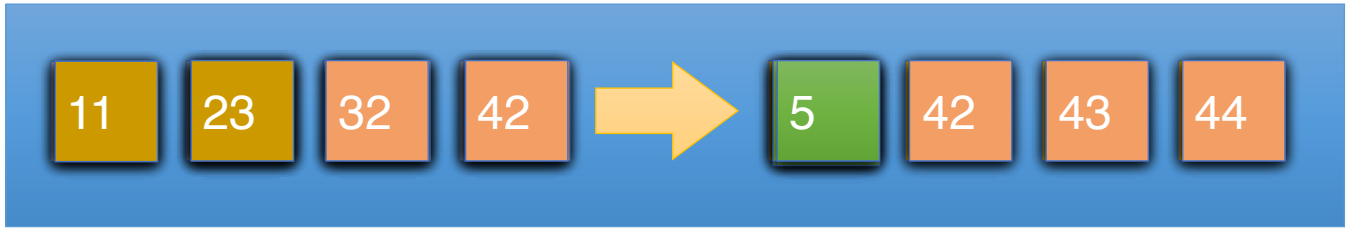
- Simplest, most intuitive join implementation

- For every k blocks of R
  - For every k' blocks of S
    - Perform all-way join across pairs of these blocks

- Variants:
  - Index-nested loop uses an index in the "inner" loop to look up only blocks of S that can match the k blocks of R
  - If one of the relations (say S) fits entirely in memory, we only need to cycle through the blocks of R in the "outer" loop

Join Approach 2: Two Pass Sorting-based Join

- Similar to the two-pass sort



- Phase 1
  - Sort R on join attrib, write out
  - Sort S on join attrib, write out
- Phase 2
  - Merge by reading through sorted R and S on join attrib

- Also known as "sort-merge" join
- Additional benefit: output is sorted

Join Approach 2: Two Pass Sorting-based Join

- Phase 1
  - Sort R on join attrib, write out
  - Sort S on join attrib, write out
- Phase 2
  - Merge by reading through sorted R and S on join attrib

- Variants:
  - Even more convenient when one of the relations is sorted already
  - An index (e.g., B+ tree) can be used to retrieve R or S in sorted order
  - Like nested-loop, can be easier if one of the relations fits in memory

Join Approach 3: Two Pass Hash-based Join

- Phase 1:
  - Hash R into buckets b1, b2, … based on join attrib
  - Hash S into buckets b1, b2, … based on join attrib
- Phase 2:
  - Read one bucket at a time, perform join

Join Approach 3: Two Pass Hash-based Join

- Phase 1:
  - Hash R into buckets b1, b2, … based on join attrib
  - Hash S into buckets b1, b2, … based on join attrib
- Phase 2:
  - Read one bucket at a time, perform join

- Variant:
  - If one of the relations (say R) fits entirely in memory, can create the hash table for R first, and then cycle through blocks of S, probe the hash table to find all the output tuples

Similar Variants Exist for Other Binary Operators

- Typically Hashing, Sorting, and Index-Nested based variants

- Exercise:
  - How would you use hashing to compute A - B?
  - How would you use sorting to compute A - B?
  - How would you use indexing to compute A - B?

  - Recall that A - B is set oriented by default for SQL

For Other Unary Operators (Apart from Sort)

- Filters/Projects can simply be "applied" to the results of an operation; no specific physical operator choices

- For Aggregation/Grouping:
  - Goal is to ensure that "groups" of tuples are processed together
  - Similar to binary operators, hashing and sorting are key techniques
  - If an index is present, can use index to retrieve in sorted order (even cheaper!)

- How might we be able to use hashing to perform an aggregation per group?
- What if the number of groups is very large?

Overall: Physical Design of Operators is Hard!

- Lots of variations, but most rely on a small set of techniques
  - Sorting, hashing, indexing, nested-looping
  - Some of these techniques produce outputs in sorted order, which may matter for subsequent ops.

- We haven't talked about what might work best given a certain setting
  - For that one needs a **cost model:** taking into account the sizes of the relations, the distributions of values, and buffer sizes