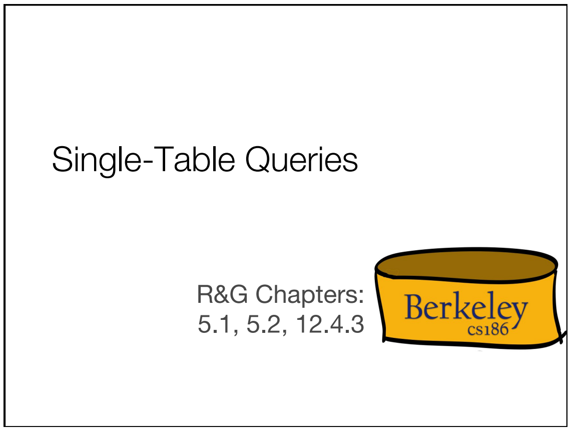
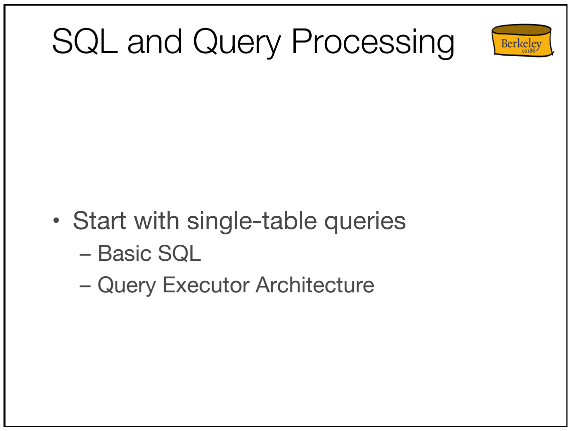


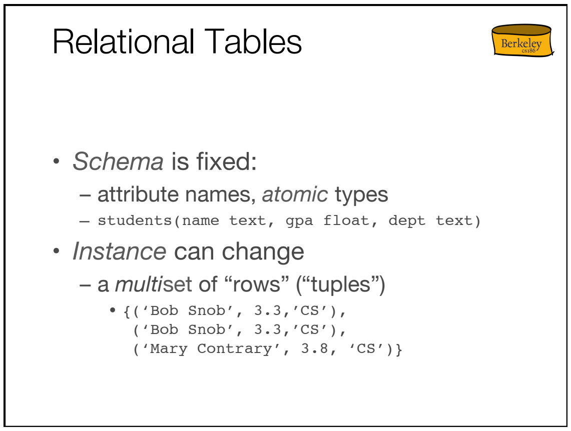
Single-Table Queries

R&G Chapters:
5.1, 5.2, 12.4.3



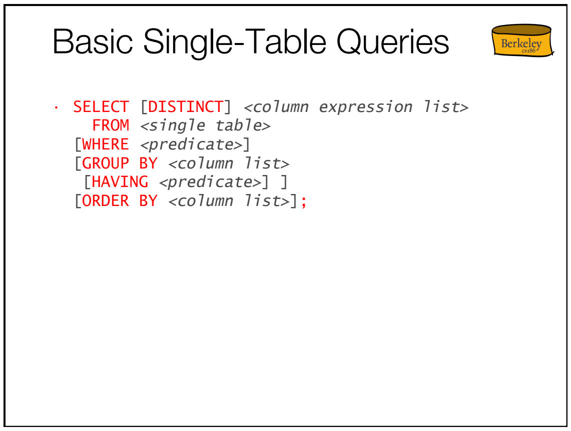
SQL and Query Processing

- Start with single-table queries
  - Basic SQL
  - Query Executor Architecture



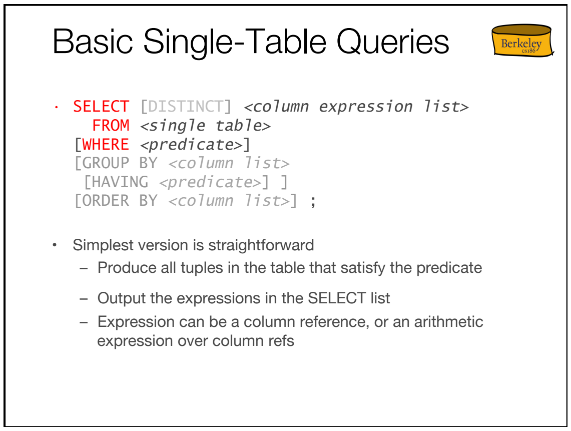
Relational Tables

- *Schema* is fixed:
  - attribute names, *atomic* types
  - `students(name text, gpa float, dept text)`
- *Instance* can change
  - a *multi*set of “rows” (“tuples”)
    - `{(‘Bob Snob’, 3.3,’CS’),`
      `(‘Bob Snob’, 3.3,’CS’),`
      `(‘Mary Contrary’, 3.8, ‘CS’)}`



Basic Single-Table Queries

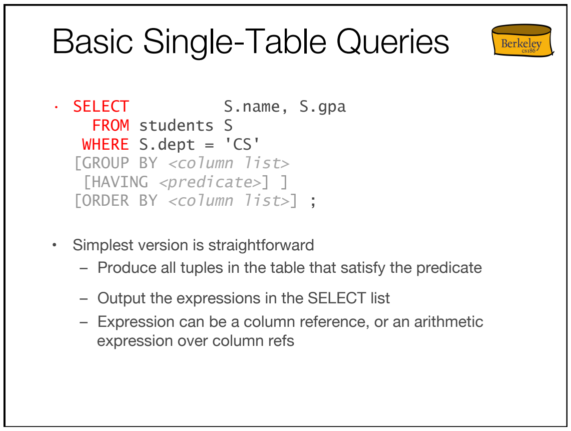
```
SELECT [DISTINCT] <column expression list>
  FROM <single table>
[WHERE <predicate>]
[GROUP BY <column list>
 [HAVING <predicate>] ]
[ORDER BY <column list>];
```



Basic Single-Table Queries

```
SELECT [DISTINCT] <column expression list>
  FROM <single table>
[WHERE <predicate>]
[GROUP BY <column list>
 [HAVING <predicate>] ]
[ORDER BY <column list>] ;
```

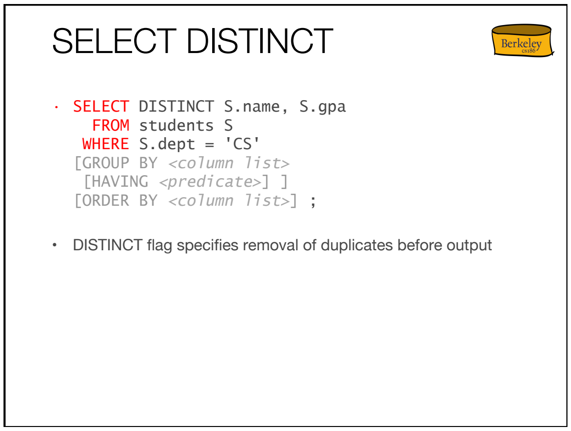
- Simplest version is straightforward
  - Produce all tuples in the table that satisfy the predicate
  - Output the expressions in the SELECT list
  - Expression can be a column reference, or an arithmetic expression over column refs



Basic Single-Table Queries

```
SELECT          S.name, S.gpa
  FROM students S
 WHERE S.dept = 'CS'
[GROUP BY <column list>
 [HAVING <predicate>] ]
[ORDER BY <column list>] ;
```

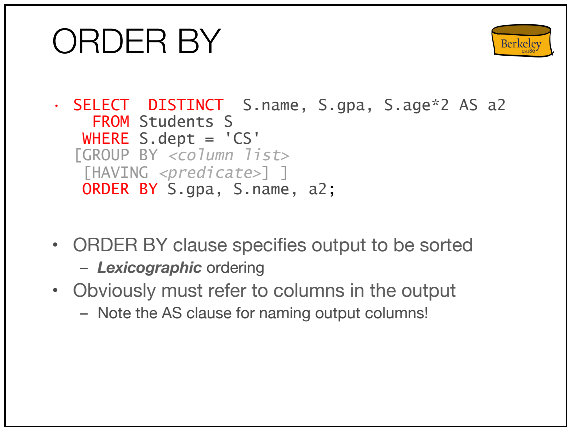
- Simplest version is straightforward
  - Produce all tuples in the table that satisfy the predicate
  - Output the expressions in the SELECT list
  - Expression can be a column reference, or an arithmetic expression over column refs



SELECT DISTINCT

```
• SELECT DISTINCT S.name, S.gpa
    FROM students S
   WHERE S.dept = 'CS'
  [GROUP BY <column list>
   [HAVING <predicate>] ]
  [ORDER BY <column list>] ;
```

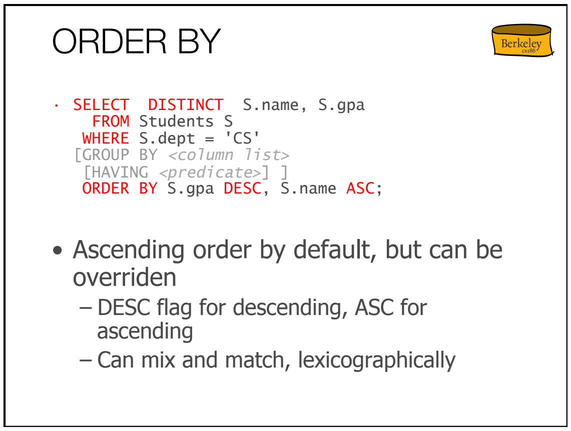
- DISTINCT flag specifies removal of duplicates before output



ORDER BY

```
• SELECT  DISTINCT  S.name, S.gpa, S.age*2 AS a2
    FROM Students S
   WHERE S.dept = 'CS'
  [GROUP BY <column list>
   [HAVING <predicate>] ]
   ORDER BY S.gpa, S.name, a2;
```

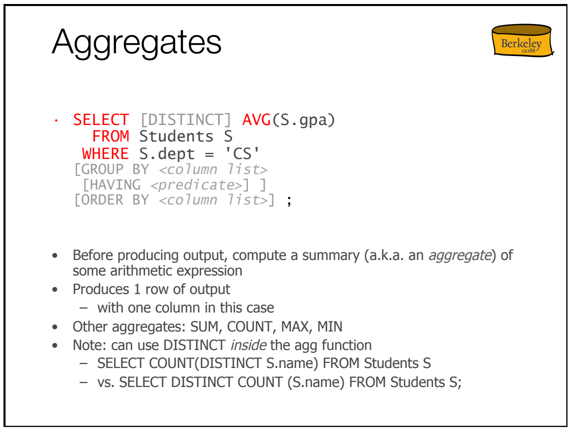
- ORDER BY clause specifies output to be sorted
  - ***Lexicographic*** ordering
- Obviously must refer to columns in the output
  - Note the AS clause for naming output columns!



ORDER BY

```
• SELECT  DISTINCT  S.name, S.gpa
    FROM Students S
   WHERE S.dept = 'CS'
  [GROUP BY <column list>
   [HAVING <predicate>] ]
   ORDER BY S.gpa DESC, S.name ASC;
```

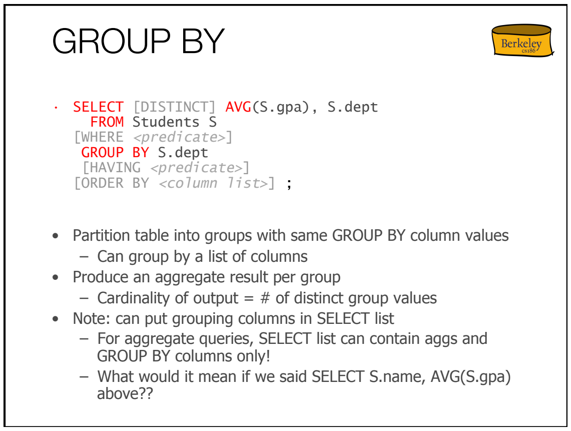
- Ascending order by default, but can be overriden
  - DESC flag for descending, ASC for ascending
  - Can mix and match, lexicographically



Aggregates

```
• SELECT [DISTINCT] AVG(S.gpa)
    FROM Students S
   WHERE S.dept = 'CS'
  [GROUP BY <column list>
   [HAVING <predicate>] ]
  [ORDER BY <column list>] ;
```

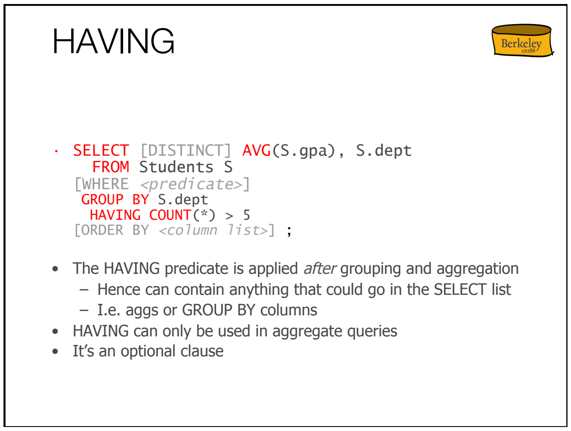
- Before producing output, compute a summary (a.k.a. an *aggregate*) of some arithmetic expression
- Produces 1 row of output
  - with one column in this case
- Other aggregates: SUM, COUNT, MAX, MIN
- Note: can use DISTINCT *inside* the agg function
  - SELECT COUNT(DISTINCT S.name) FROM Students S
  - vs. SELECT DISTINCT COUNT (S.name) FROM Students S;



GROUP BY

```
• SELECT [DISTINCT] AVG(S.gpa), S.dept
    FROM Students S
  [WHERE <predicate>]
   GROUP BY S.dept
   [HAVING <predicate>]
  [ORDER BY <column list>] ;
```

- Partition table into groups with same GROUP BY column values
  - Can group by a list of columns
- Produce an aggregate result per group
  - Cardinality of output = # of distinct group values
- Note: can put grouping columns in SELECT list
  - For aggregate queries, SELECT list can contain aggs and GROUP BY columns only!
  - What would it mean if we said SELECT S.name, AVG(S.gpa) above??



HAVING

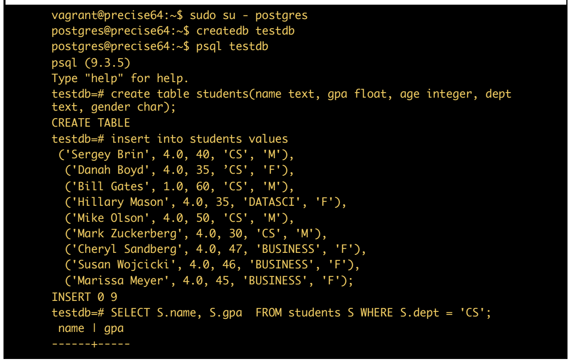
```
• SELECT [DISTINCT] AVG(S.gpa), S.dept
    FROM Students S
  [WHERE <predicate>]
   GROUP BY S.dept
    HAVING COUNT(*) > 5
  [ORDER BY <column list>] ;
```

- The HAVING predicate is applied *after* grouping and aggregation
  - Hence can contain anything that could go in the SELECT list
  - I.e. aggs or GROUP BY columns
- HAVING can only be used in aggregate queries
- It's an optional clause

Putting it all together

```
· SELECT S.dept, AVG(S.gpa), COUNT(*)
     FROM Students S
    WHERE S.gender = 'F'
    GROUP BY S.dept
  HAVING COUNT(*) > 5
    ORDER BY S.dept;
```

Try It Yourself

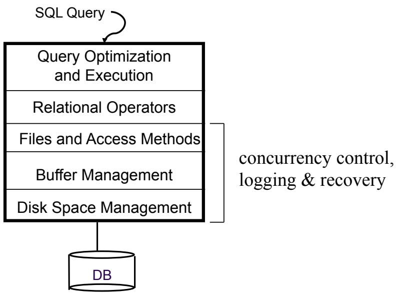


```
vagrant@precise64:~$ sudo su - postgres
postgres@precise64:~$ createdb testdb
postgres@precise64:~$ psql testdb
psql (9.3.5)
Type "help" for help.
testdb=# create table students(name text, gpa float, age integer, dept
text, gender char);
CREATE TABLE
testdb=# insert into students values
 ('Sergey Brin', 4.0, 40, 'CS', 'M'),
  ('Danah Boyd', 4.0, 35, 'CS', 'F'),
  ('Bill Gates', 1.0, 60, 'CS', 'M'),
  ('Hillary Mason', 4.0, 35, 'DATASCI', 'F'),
  ('Mike Olson', 4.0, 50, 'CS', 'M'),
  ('Mark Zuckerberg', 4.0, 30, 'CS', 'M'),
  ('Cheryl Sandberg', 4.0, 47, 'BUSINESS', 'F'),
  ('Susan Wojcicki', 4.0, 46, 'BUSINESS', 'F'),
  ('Marissa Meyer', 4.0, 45, 'BUSINESS', 'F');
INSERT 0 9
testdb=# SELECT S.name, S.gpa  FROM students S WHERE S.dept = 'CS';
 name | gpa
------+-----
```

Context

- We looked at SQL
- Now shift gears and look at SW architecture for DBMS query processing

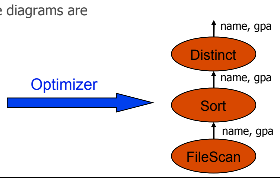
Typical DBMS architecture



Query Processing Overview

- The *query optimizer* translates SQL to a special internal "language"
  - Query Plans
- The *query executor* is an *interpreter* for query plans
- Think of query plans as "blobs-and-arrows" *dataflow* diagrams
  - Each blob implements a *relational operator*
  - Edges represent a flow of tuples (columns as specified)
  - For single-table queries, these diagrams are straight-line graphs

```
SELECT DISTINCT name, gpa
  FROM Students
```



Iterators

iterator

- The relational operators are all subclasses of the class `iterator`:

```
class iterator {
    void init();
    tuple next();
    void close();
    iterator &inputs[];
    // additional state goes here
}
```

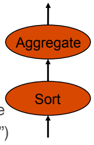
- Note:
  - Edges in the graph are specified by inputs (max 2, usually)
  - Encapsulation: any iterator can be input to any other!
  - When subclassing, different iterators will keep different kinds of state information

Example: Sort

```
class Sort extends iterator {
    void init();
    tuple next();
    void close();
    iterator &inputs[1];
    int numberOfRuns;
    DiskBlock runs[];
    RID nextRID[];
}
```

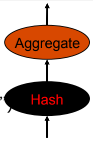
- `init()`:
  - generate the sorted runs on disk
  - Allocate `runs[]` array and fill in with disk pointers.
  - Initialize numberOfRuns
  - Allocate nextRID array and initialize to NULLs
- `next()`:
  - nextRID array tells us where we’re “up to” in each run
  - find the next tuple to return based on nextRID array
  - advance the corresponding nextRID entry
  - return tuple (or EOF -- “End of Fun” -- if no tuples remain)
- `close()`:
  - deallocate the runs and nextRID arrays

Sort GROUP BY



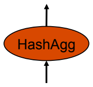
- The Sort iterator ensures all its tuples are output in sequence
- The Aggregate iterator keeps running info (“transition values”) on agg functions in the SELECT list, per group
  - E.g., for COUNT, it keeps `count-so-far`
  - For SUM, it keeps `sum-so-far`
  - For AVG it keeps `sum-so-far` *and* `count-so-far`
- As soon as the Aggregate iterator sees a tuple from a new group:
  1. It produces output for the old group based on agg function
     E.g. for AVG it returns (`sum-so-far/count-so-far`)
  2. It resets its running info.
  3. It updates the running info with the new tuple’s info

Hash GROUP BY (naïve)

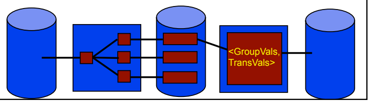


- The Hash iterator ensures all its tuples are output in batches
- The Aggregate iterator keeps running info (“transition values”) on agg functions in the SELECT list, per group
  - E.g., for COUNT, it keeps `count-so-far`
  - For SUM, it keeps `sum-so-far`
  - For AVG it keeps `sum-so-far` *and* `count-so-far`
- As soon as the Aggregate iterator sees a tuple from a new group:
  1. It produces output for the old group based on agg function
     E.g. for AVG it returns (`sum-so-far/count-so-far`)
  2. It resets its running info.
  3. It updates the running info with the new tuple’s info

We Can Do Better!



- Combine the summarization into the hashing process
  - During the ReHash phase, don’t store tuples, store pairs of the form <GroupVals, TransVals>
  - When we want to insert a new tuple into the hash table
    - If we find a matching GroupVals, just update the TransVals appropriately
    - Else insert a new <GroupVals,TransVals> pair
- What’s the benefit?
  - Q: How many pairs will we have to hash?
  - A: Number of distinct values of GroupVals columns
    - Not the number of tuples!!
  - Also probably “narrower” than the tuples



Summary

- Intro to SQL aggregation, etc.
- Iterator architecture of a query executor
  - Streams data through operators by “pulling”
    - Lazy evaluation
  - Encapsulates operator logic
    - Arbitrary dataflow composition