Spring 2024 CSE 380

2-1-2024

Quiz Info

- Quiz 2 is next Tuesday, February 6th

Review

Foreign Keys

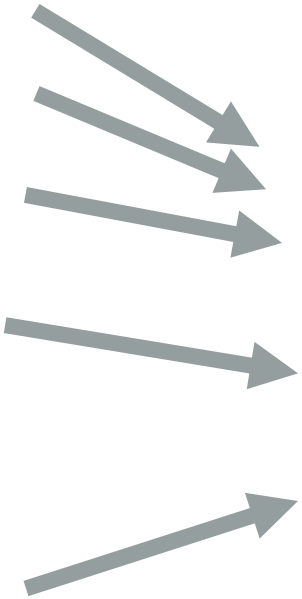
- We discussed primary keys, which are columns that uniquely identify each row.
- However, often our tables will have columns that are meant to match up with columns in a different table.
- We want to add a constraint on those columns that there must be an associated row in a foreign (other) table.
  - NULLs are okay

More Table Constraints

tracks

id	name	artist_id
1	Let it be	1
2	Penny Lane	1
3	Yellow Submarine	1
4	Hedwig's Theme	2
5	Jingle Bell Rock	NULL
6	The Devil Is A Patient Man	3

tracks.artist_id is a foreign key referencing artists.id



artists

id	name
1	The Beatles
2	John Williams
3	CRUD

Example

```
PRAGMA foreign_keys = ON;
INSERT INTO artists (id, name) VALUES (1, 'Beatles');
INSERT INTO tracks (id, name, artist_id) VALUES
       (1, 'Jingle Bell Rock', NULL), -- OKAY
       (2, 'Let it be', 1), -- Okay
       (3, 'Jurassic Park', 2); -- ERROR: no matching key
```

Parameterized Queries

```
conn = sqlite3.connect(":memory:")
conn.execute("CREATE TABLE students (name TEXT, age INTEGER);")
conn.execute("INSERT INTO students VALUES ('James', 30);")
```

What if we wanted to add a python integer?

```
Steve_age = 23
conn.execute( "INSERT INTO students VALUES ('Steve', " + str(steve_age) + ");" )
```

Parameterized Queries

- Used to pass python objects into queries without needing to manually convert to strings.

```
Steve_age = 23
conn.execute("INSERT INTO students VALUES ('Steve', ?);", (steve_age,))
row = ('Tim', 45)
conn.execute("INSERT INTO students VALUES (?, ?);", row)
```

Protection by Parameterized Query

- Parameterized queries will automatically escape the input and ensure that the value passed in is store as that value.

```
name = "Robert'); DROP TABLE students; --"
conn.execute("INSERT INTO students VALUES (?);", (name,))
```

- The string name will be stored, in its entirety, and the single quote will be escaped to stop it from harming the database.

Python and SQLite

Example Code

- This will be posted online

```
import sqlite3
conn = sqlite3.connect("test.db")
curr = conn.cursor()

curr.execute("DROP TABLE IF EXISTS students")
curr.execute("CREATE TABLE students (col1 INTEGER, col2 TEXT, col3 REAL);")

curr.execute("INSERT INTO students VALUES (3, 'hi', 4.5);")

multiple_records = [(7842, 'string with spaces', 3.0), (7, 'look a null', None)]
curr.executemany("INSERT INTO students VALUES (?, ?, ?);", multiple_records)

curr.execute("SELECT col1, col2, col3 FROM students ORDER BY col1;")
result_list = curr.fetchall() #fetchone(), fetchmany(3)

expected = [(3, 'hi', 4.5), (7, 'look a null', None), (7842, 'string with spaces', 3.0)]

print("expected:",  expected)
print("actual: ",  result_list)
assert expected == result_list
```

New Material

Functions

- Functions take 0 or more arguments and return one result. Here are some more:
  - abs(x) - returns the absolute value of its argument
  - lower(x) - returns a copy of a string in lower case
  - upper(x) - guess what this does
  - random(x) - returns an random, signed 64-bit integer
  - typeof(x) - returns the type ("null", "integer", "real", "text", "blob") of x
  - round(x, y) - returns a floating-point value x rounds to y digits after the decimal point
  - trim(x, y) - returns a copy of string x with y characters removed from each end

User Defined Functions

- What if we wanted to add a custom function to SQLite?
  - title(x) - returns the string x in "Title Case" (the first letter of each word is capitalized).
- We first write a function (in Python) that does what we want:

```
def make_title(x):
  return x.title()
```

- Then we need to tell our connection to the database that this function exists:

```
conn.create_function("title", 1, make_title)
```

- Now we can use it in SQL queries:

```
curr.execute("SELECT title(name) FROM students;")
```

create_function

- The create_function method on the connection object takes three parameters:
  - name - a string giving the name the function will be called by in SQL
  - num_params - the number of parameters the function requires
  - You can overload functions with different number of arguments
  - func - the function (in Python) to be called
- The function can return any of the SQLite supported types (bytes, str, int, float, and None)

Aggregate Functions

- Aggregators (aggregate functions) take 0 or more values and return some form of summary or those values. When seen many before:
  - count(x) - returns the number of items in x
  - max(x) - returns the largest value in x
  - ...
- You can also create a custom aggregator with the Python sqlite3 module
  - Lets say we wanted an aggregate function called char_length that tallied the number of characters in the strings passed to it
- Example Usage:

```
CREATE TABLE x (col TEXT);
INSERT INTO x VALUES ('hi'), ('bye');
SELECT char_length(col) FROM x; -- returns 5
```

Custom Aggregator

```
class CharCounter:
  def __init__(self):
    self.char_count = 0
  def step(self, value):
    self.char_count += len(value)
  def finalize(self):
    return self.char_count
conn.create_aggregate("char_length", 1, CharCounter)
```

create_aggregate

- Just like create_function, the create_aggregate method of the SQLite3 Connection class has three arguments:
  - name - the SQL name to call the aggregate function by
  - num_params - the number of parameters for the aggregate function
  - aggregate_class - the Python class to pass data to
- The aggregate class must have three methods:
  - __init__(self) - the init method that initializes the class (usually an attribute to 0)
  - step(self, value) - a method that is called for every values to be aggregated
  - finalize(self) - a method that returns the value for the aggregate function/class

Storing Custom Types

- Often you want to store different types of data in a database. The database can only hold a few types of data (INTEGER, REAL, TEXT, BLOB, NULL), but you can use adapters and converters to change objects into these few types and back.
- Adapter - A function to convert a Python type into one of SQL's supported types (string or bytestring)
- Converter - A function to convert a bytestring (BLOB) to a Python type.
- We've can use these whenever we store Python datetime objects in SQLite.

Custom Python Type

- Lets say we wanted to represent colors in our database. We have a Color class in Python:

```
class Color:
  def __init__(self, r, g, b):
    self.r, self.g, self.b = r, g, b
  def __repr__(self): return "Color({}, {}, {})".format(
    self.r, self.g, self.b)
```

- Then we need two functions (an adapter and a converter) to convert to a form that can be put into a Sqlite database and returned to being an Python object.

Adapter and Converter

```
def adapt_color(color):
  return "{};{};{}".format(color.r, color.g, color.b)
def convert_color(bytestring):
  as_str = bytestring.decode('ascii')
  r, g, b = [float(x) for x in as_str.split(';')]
  return Color(r, g, b)

sqlite3.register_adapter(Color, adapt_color)
sqlite3.register_converter("COLOR", convert_color)
```

Python Types

- Adapters take a custom python object (like a "Color" object), and return a string containing the data from that object.
- Converters take a bytes object (python binary string) and should return a new python object instance.
- You can convert a bytes object to the more familiar python strings with:
  - as_str = as_bytes.decode('ascii')
  - Now as_str holds a python string that you can split and do other things with.

Using with PARSE_DECLTYPES

```
c = Color(24, 69, 59)
conn = sqlite3.connect(":memory:", detect_types=sqlite3.PARSE_DECLTYPES)
curr = conn.cursor()
curr.execute("CREATE TABLE test(col COLOR);")
curr.execute("INSERT INTO test(col) VALUES(?);", (c,))
curr.execute("SELECT col FROM test;")
row = curr.fetchall()
row = next(res)
print("with declared types:", row[0])
```

"detect_types=sqlite3.PARSE_DECLTYPES" instructs SQLite to try to convert values inserted into a column with a custom type to the associated Python type.

Views

- Views are named SELECT Statement

students

id	name
1	James
2	Abigail

log

student_id	time_inserted
1	12:40:00
2	14:37:34

```
CREATE VIEW name_and_time AS
        SELECT students.name, log.time_inserted FROM students
                INNER JOIN log ON students.id = log.student_id;
```

name_and_time

name	time_inserted
James	12:40:00
Abigail	14:37:34

Views Cont.

- A view is a virtual table composed of the result-set of a select statement.
- A view has rows and columns (just like a table), but instead of holding data itself, it just points to data held in other tables.
- You can treat a view like a table and do select statements on it,
  - But views cannot be modified (no insert, update or delete)
- You can use a view like a temporary table. You can build the view from multiple other tables with joins and/or filter certain rows with WHERE and so on.
- If the view's SELECT statement changes (modifications where made to the SELECT's underlying tables), the view automatically changes too.

View Example

```
CREATE VIEW artist_and_albums AS
        SELECT Artist.Name, Album.Title, Album.AlbumId
        FROM Artist INNER JOIN Album ON Artist.ArtistId = Album.ArtistId;
CREATE VIEW good_artists_and_albums AS
        SELECT * FROM artist_and_album WHERE NOT Name LIKE '%Taylor Swift%';
CREATE VIEW good_tracks AS
        SELECT Tracks.* FROM Tracks INNER JOIN good_artists_and_albums
        ON Tracks.AlbumId = good_artists_and_albums.AlbumId;
```

When Should you use Views?

1. When you want a read-only table made from data in other tables.
2. To hide data complexity (for instance, complicated joins).
3. Customizing the data, using functions and group by, to present the data in a new form.
4. Security and privacy

- However, views are as slow as the query that creates them. And, views of views are even slower as they are basically nested SELECT subqueries.

View Lifespan

- A view lives only as long as the tables in its select statement.
  - If the tables in the select statement are dropped, the view is also dropped.
- You can use the TEMPORARY keyword to indicate that the view should automatically be dropped when the current database connection is closed (meaning the view isn't persistent across connections).

CREATE TEMPORARY VIEW something AS SELECT * FROM students;

You can also remove a view manually with DROP VIEW:

DROP VIEW something;

Indices

Note: Indices not on Quiz 2

- What is an index?
  - Like the index in the back of a book (remember those?), an index helps you find information quickly.
  - Without an index, you would have to go through the entire book to find a particular topic.
    - But with an index, you can look up the topic in a sorted area, and then jump to the topic you care about.
  - Indices have no external effect on a database; they do not affect any of the other queries.
    - But they do have a huge effect on performance.
  - By default, every table is indexed according to its integer primary key. If there is no primary key, SQLite makes a hidden one called "rowid".

Create Index

CREATE INDEX index_name ON students (name);

- You can add an index to a database with the above command

CREATE INDEX index_name students (id, name);

- You can have multiple columns to sort your index on.

DROP INDEX index_name;

DROP INDEX IF EXISTS index_name;

- As always IF [NOT] EXISTS makes the CREATE/DROP a no-op if an table/view/index with that name already exists

When Should you Create an Index?

- When you need to often look up rows, but not according to their primary key.
- An index on some columns in a table allows it to quickly get matching rows when doing a lookup on those columns.
- But, indices aren't always good, modifications to a table (UPDATE, INSERT, DELETE) all need to adjust the indices on the table, leading to slower performance for those operations.
- Also, indices take up memory, so large/multiple indices can take up valuable space in your database.

Query Planner

- The following content is taken from https://www.sqlite.org/queryplanner.html.
- SQL is a declarative language, not a procedural language, meaning you write what you want to do, not how to do it.
- It is the job of the SQL implementation to interpret your declaration into actual steps to yield the answer.
- Most of the time the implementation can do this efficiently, but sometimes it needs you to create indices to make the operations you need done faster.

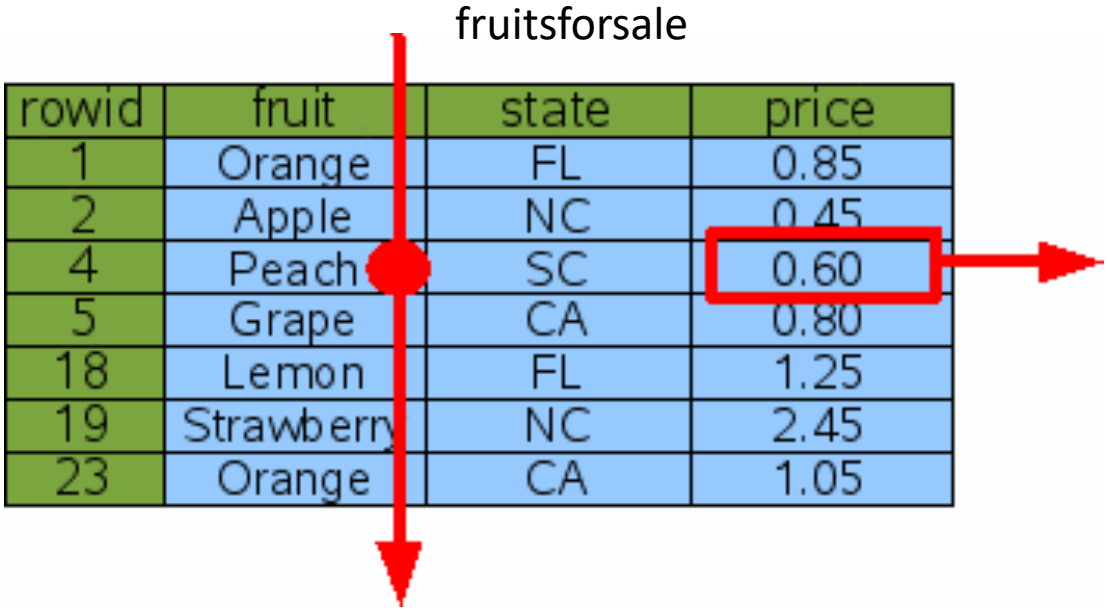
```
CREATE TABLE
fruitsforsale (
        fruit TEXT,
        state TEXT,
        price REAL
);
INSERT INTO ...
```

fruitsforsale

rowid	fruit	state	price
1	Orange	FL	0.85
2	Apple	NC	0.45
4	Peach	SC	0.60
5	Grape	CA	0.80
18	Lemon	FL	1.25
19	Strawberry	NC	2.45
23	Orange	CA	1.05

We didn't specify a INTEGER PRIMARY KEY, so a new column "rowid" was made for us as the INTEGER PRIMARY KEY.
The table is always ordered according to the key.

```
SELECT price FROM
fruitsforsale WHERE
fruit = 'Peach';
```



Without an index on fruit, we have to do a full table scan, meaning we must examine every row and check if fruit is equal to 'Peach'.

This is very slow if you have thousands or millions of rows.

What is the Big O Notation for a Simple SELECT Statement? (n is number of rows)

- n
- log(n)
- log(n) + log(n)
- Uhhhh...

```
CREATE INDEX
fruit_index ON
fruitsforsale(fruit);
```

Now we have a index ordered by fruit so we can quickly find rows with a certain fruit value.

fruit_index

fruit	rowid
Apple	2
Grape	5
Lemon	18
Orange	1
Orange	23
Peach	4
Strawberry	19

```
SELECT price FROM fruitsforsale WHERE fruit = 'Peach';
```

fruit_index

fruit	rowid
Apple	2
Grape	5
Lemon	18
Orange	1
Orange	23
Peach	4
Strawberry	19

fruitsforsale

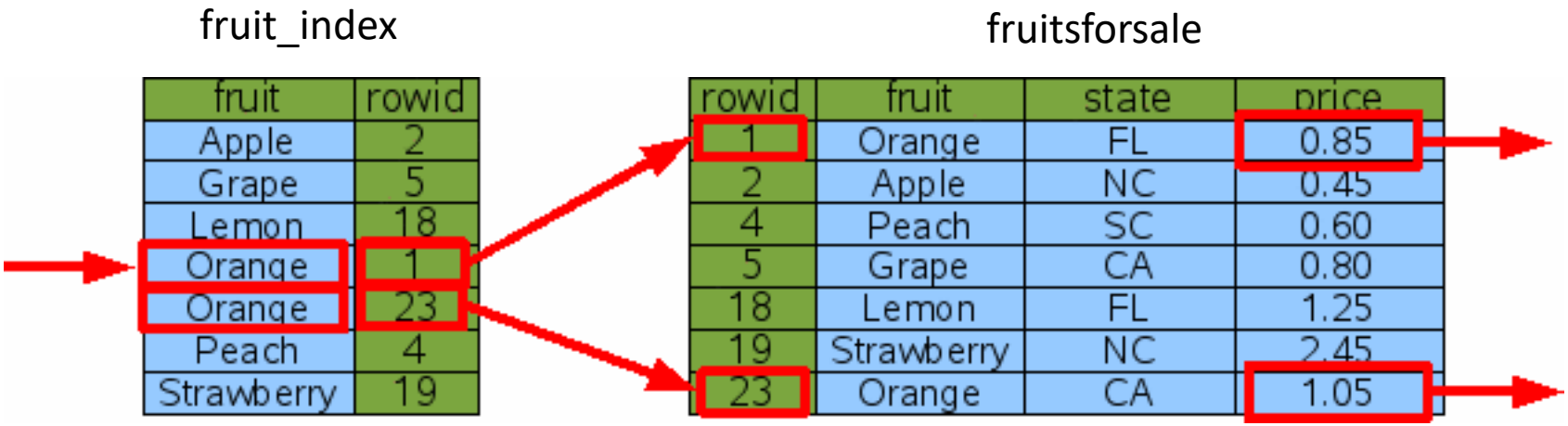
rowid	fruit	state	price
1	Orange	FL	0.85
2	Apple	NC	0.45
4	Peach	SC	0.60
5	Grape	CA	0.80
18	Lemon	FL	1.25
19	Strawberry	NC	2.45
23	Orange	CA	1.05

1. Do a binary search in fruit_index for 'Peach', then we find the rowid.
2. Do a binary search in fruitsforsale for that rowid, then we can return the price.

What is the Big O Notation for a Lookup with an Index?

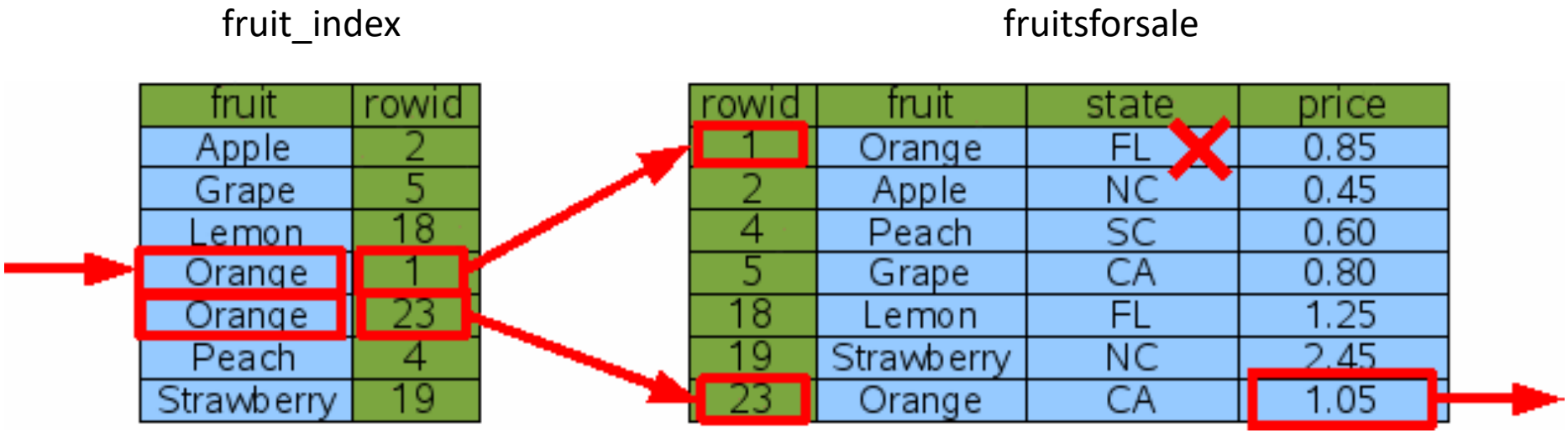
- n
- log(n)
- log(n) + log(n)
- Uhhhh...

```
SELECT price FROM fruitsforsale WHERE fruit = 'Orange';
```



1. Do a binary search in fruit_index for 'Orange', then we find the rowid.
2. Do a binary search in fruitsforsale for that rowid, then we can return the price.
3. Go back to fruit_index and check if the next row is also an orange, if so look up its row in fruitsforsale.
4. Repeat until there are no more 'Orange's.

```
SELECT price FROM fruitsforsale WHERE fruit = 'Orange' AND
state = 'CA';
```



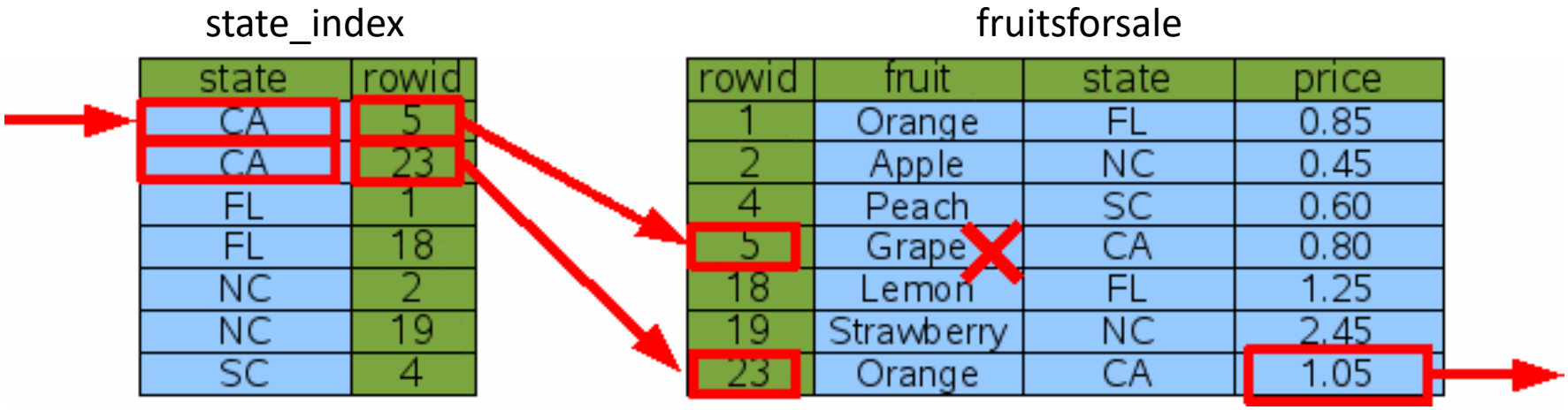
Same process as before, but we have to exclude some of the 'Orange' rows if the state doesn't match 'CA'.

```
CREATE INDEX state_index ON fruitsforsale(state);
```

state_index

state	rowid
CA	5
CA	23
FL	1
FL	18
NC	2
NC	19
SC	4

```
SELECT price FROM fruitsforsale WHERE fruit =
'Orange' AND state = 'CA';
```



Using the state_index instead of the fruit_index follows the same process.

If you have Multiple Indices, Which one is Faster?

- The index with the most rows
- The index with the least duplicates
- The index that is sorted
- Depends

```
CREATE INDEX fruit_state_index ON fruitsforsale(fruit,
state);
```

fruit_state_index

fruit	state	rowid
Apple	NC	2
Grape	CA	5
Lemon	FL	18
Orange	CA	23
Orange	FL	1
Peach	SC	4
Strawberry	NC	19

Multi-column index that is sorted according to the first column (ties broken by subsequent columns.

```
SELECT price FROM fruitsforsale WHERE
fruit = 'Orange' AND state = 'CA';
```

fruit_state_index

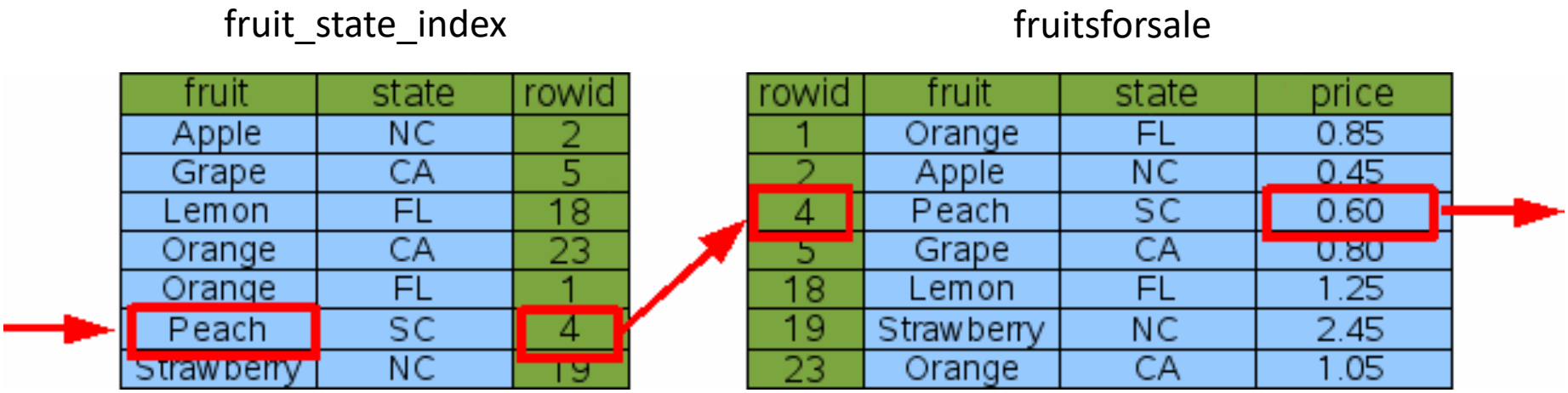
fruit	state	rowid
Apple	NC	2
Grape	CA	5
Lemon	FL	18
Orange	CA	23
Orange	FL	1
Peach	SC	4
Strawberry	NC	19

fruitsforsale

rowid	fruit	state	price
1	Orange	FL	0.85
2	Apple	NC	0.45
4	Peach	SC	0.60
5	Grape	CA	0.80
18	Lemon	FL	1.25
19	Strawberry	NC	2.45
23	Orange	CA	1.05

Using the fruit_state_index allows only finding the rows we want.

```
SELECT price FROM fruitsforsale WHERE fruit
= 'Peach';
```



fruit_state_index has all of utility fruit_index had, we can just ignore the state if it isn't needed.

```
CREATE INDEX fruit_state_price_index ON
fruitsforsale(fruit, state, price);
```

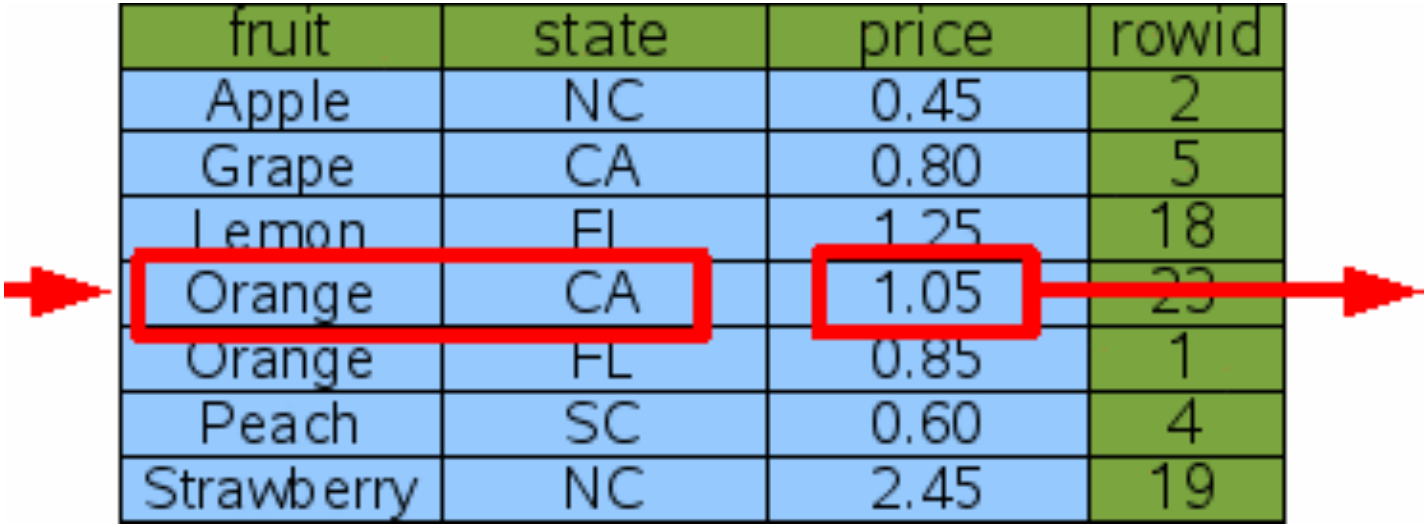
fruit_state_price_index

fruit	state	price	rowid
Apple	NC	0.45	2
Grape	CA	0.80	5
Lemon	FL	1.25	18
Orange	CA	1.05	23
Orange	FL	0.85	1
Peach	SC	0.60	4
Strawberry	NC	2.45	19

This is called a covering index - it has all of the columns used in the SELECT statement, including the output ('price').

```
SELECT price FROM fruitsforsale WHERE
fruit='Orange' AND states = 'CA';
```

fruit_state_price_index



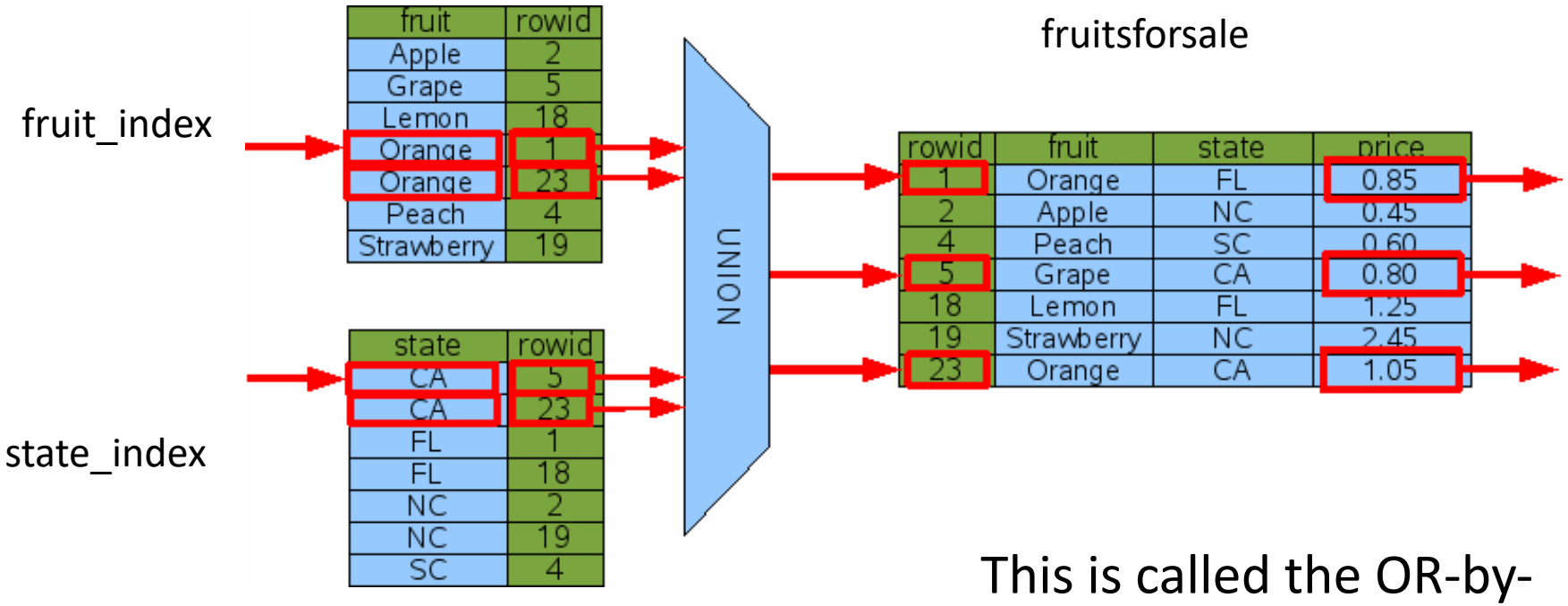
fruit	state	price	rowid
Apple	NC	0.45	2
Grape	CA	0.80	5
Lemon	FL	1.25	18
Orange	CA	1.05	23
Orange	FL	0.85	1
Peach	SC	0.60	4
Strawberry	NC	2.45	19

A covering index for a query doesn't have to consult the original table, because all of the information is in the index.
This means the second binary lookup isn't done.

What is the Big O Notation for a Lookup with a Covering Index?

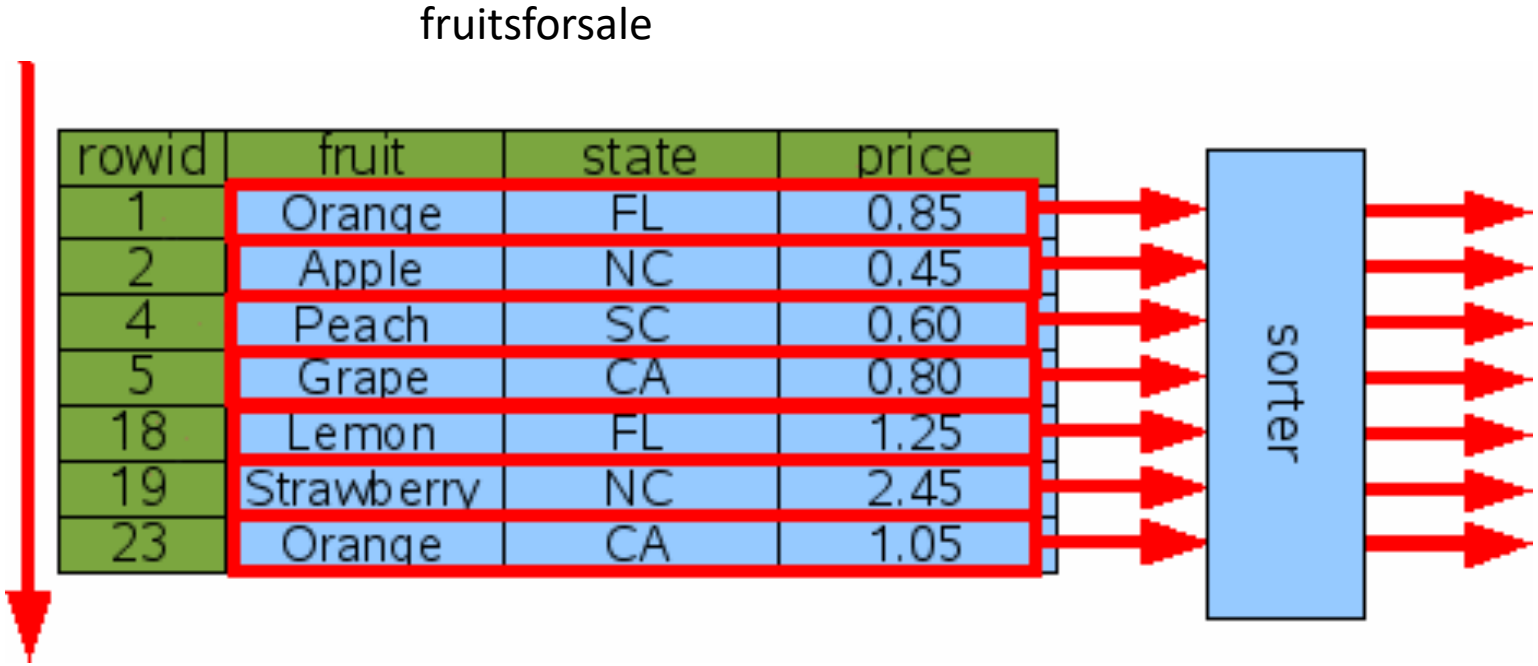
- n
- log(n)
- log(n) + log(n)
- Uhhhh...

```
SELECT price FROM fruitsforsale WHERE
fruit='Orange' OR state='CA';
```



This is called the OR-by-UNION technique.

```
SELECT * FROM fruitsforsale ORDER BY fruit;
```

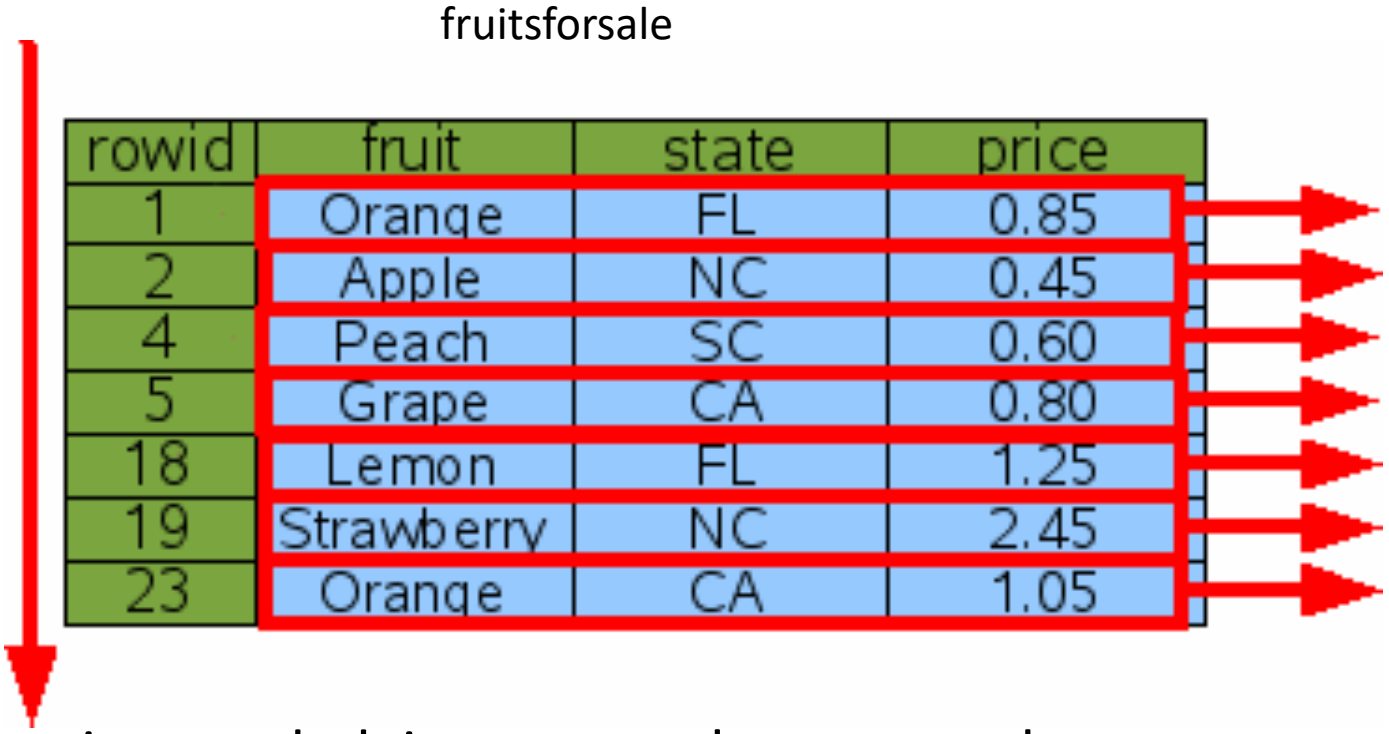


If we didn't have an index, every row is passed to a sorter function.

What is the Big O Notation for Sorting Without an Index?

- n
- log(n)
- n log(n)
- Uhhhh...

```
SELECT * FROM fruitsforsale ORDER BY rowid;
```

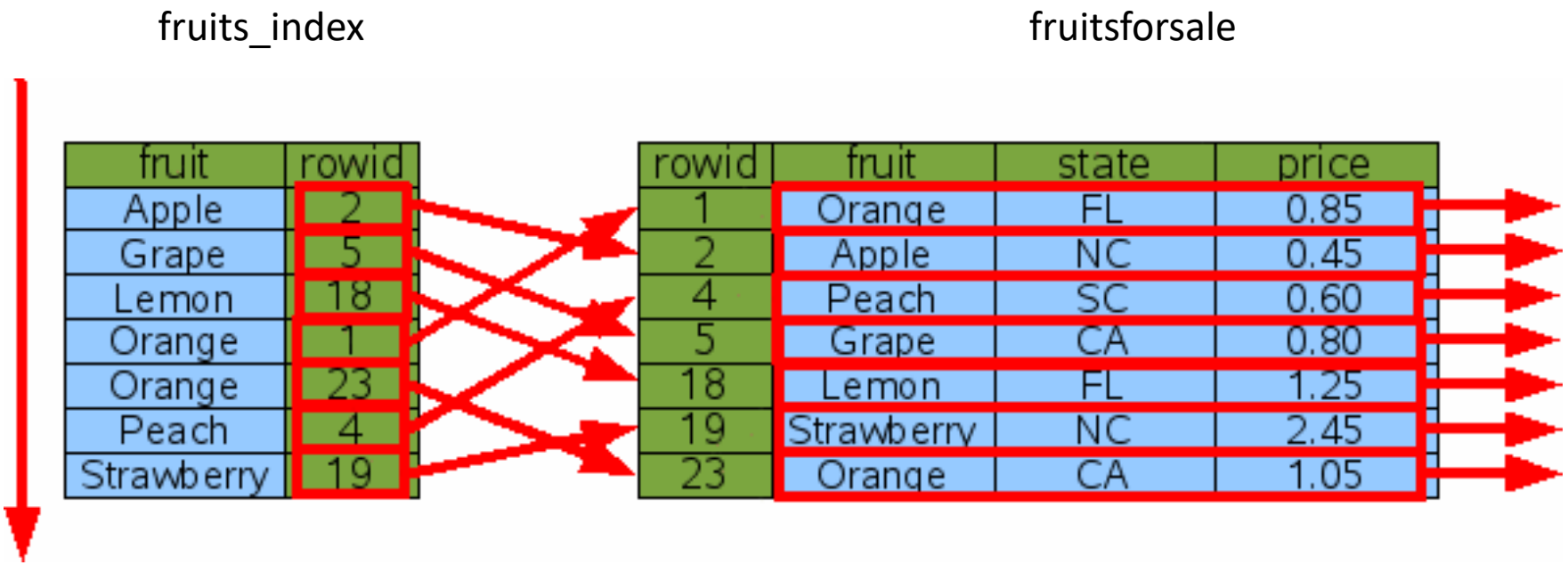


No sorting needed, just return the rows as they are.

Can you imagine what happens with:

"`SELECT * FROM fruitsforsale ORDER BY rowid DESC;`"

```
SELECT * FROM fruitsforsale ORDER BY fruit;
```

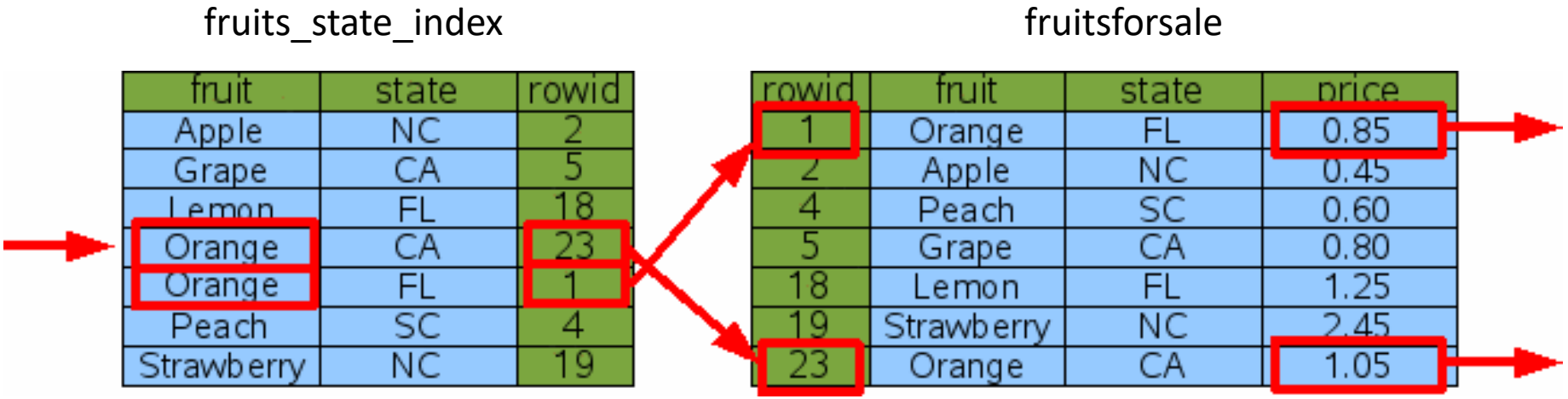


Using the index, we don't have to pass everything to a sorted function

What is the Big O Notation for Sorting with an Index?

- n
- log(n)
- n log(n)
- Uhhhh...

```
SELECT price FROM fruitsforsale WHERE
fruit='Orange' ORDER BY state;
```

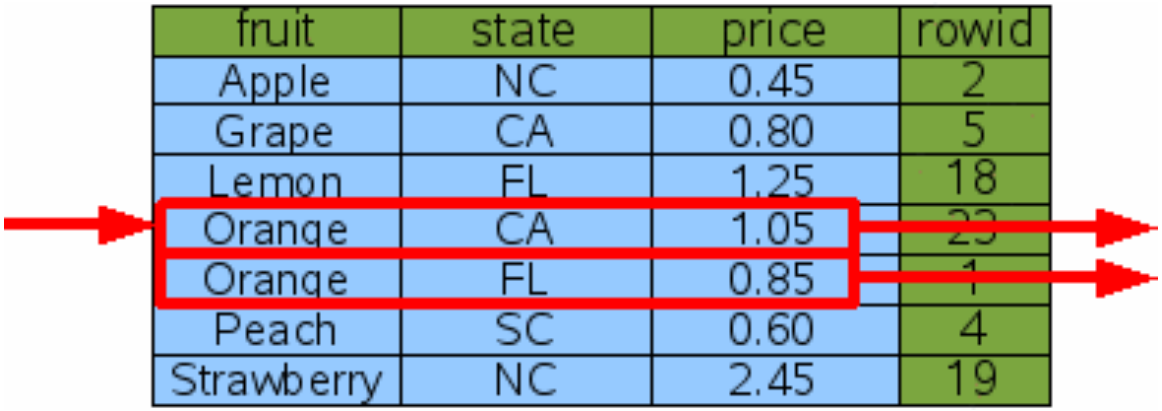


If we are sorting and filtering, using a appropriate index is very fast.

1. Find rows with fruit='Orange' (left most column in index)
2. Output the price for each record (they are already sorted by state).

```
SELECT * FROM fruitsforsale WHERE
fruit='Orange' ORDER BY state;
```

fruits_state_price_index



fruit	state	price	rowid
Apple	NC	0.45	2
Grape	CA	0.80	5
Lemon	FL	1.25	18
Orange	CA	1.05	23
Orange	FL	0.85	1
Peach	SC	0.60	4
Strawberry	NC	2.45	19

Using the appropriate covering index, we don't even have to do a lookup on the underlying table.

That's it for today

- Questions?