ISTM 6212 - Week 6
More SQL, performance, indexes, ETL

Daniel Chudnov, dchud@gwu.edu

2016-10-04

Agenda

- Project 01 follow up
- Algorithm analysis
- RDBMS indexes, query analysis, and optimization
- Basic ETL in SQL
- SQL GROUPING SETS, CUBE, ROLLUP
- Exercise 04

Project 01 follow up

Common mistakes

- "`wc -w`" without prior word split
- "`grep Jo`" also matches "John"
- filter ordering
- whitespace handling in filters
- not acknowledging unexpected results (counts differ)

For future you

- Use text/Markdown cells for narrative comments
- Use Markdown formatting for document structure
- Let the notebook fetch files; don't commit them
- One cell per computation, esp. w/output
- Meaningful filenames
- No spaces in filenames

For future you

- Start early
- Run into problems early
- Come to office hours!

Subtle issues to note

- csvsort vs. sort
- csvlook on non-CSV data
- don't just copy-and-paste, cf. "`cp 2016q1.csv q4.csv`"
- "`grep -oE '\w{2,}' | ./split.py`"
- "list" and "file" are reserved words
- if #10 is a tie, show more

loops: while vs. if

a

```
while word not in stopwords:
  print(word)
  break
```

b

```
if word not in stopwords:
  print(word)
```

loops: counting steps

a

```
for word in line.split():
  stopwords = ['a', 'an', 'the', ...]
  if word in stopwords:
    print(line)
```

b

```
stopwords = ['a', 'an', 'the', ...]
for word in line.split():
  if word in stopwords:
    print(line)
```

Algorithm analysis

Algorithm analysis

- Theoretical summary of algorithm performance
- aka "complexity analysis"
- Widely used
- Important when scaling up data processing:
  - How will algorithm perform w/2x, 10x, 100x data?
- "Big-O" notation of worst case performance

Algorithm analysis - basics

```
var M = A[ 0 ];
for ( var i = 0; i < n; ++i ) {
  if ( A[ i ] >= M ) {
    M = A[ i ];
  }
}
```

example taken from http://discrete.gr/complexity/

Algorithm analysis - basics

# instructions	
2	`var M = A[ 0 ];`
2 + 2n	`for ( var i = 0; i < n; ++i ) {`
2n	`  if ( A[ i ] >= M ) {`
2n	`    M = A[ i ];`
	`  }`
	`}`

Total # instructions (worst-case): 6n + 4

Algorithm analysis - basics

- starting with $f(n) = 6n + 4$
  - remove 4, it doesn't grow with n
  - remove 6, this constant is less important than exponent of n
  - complexity is *n*, or *O(n)*, aka "big-O of n" or "linear"

loops: counting steps

a

```
for word in line.split():
  stopwords = ['a', 'an', 'the', ...]
  if word in stopwords:
    print(line)
```

b

```
stopwords = ['a', 'an', 'the', ...]
for word in line.split():
  if word in stopwords:
    print(line)
```

Algorithm analysis - nested loop

```
var M = A[ 0 ];
for ( var i = 0; i < n; ++i ) {
  for (var j = 0; j < n; ++j) {
    if ( A[ i ] >= j * M ) {
      M = A[ i ];
}}};
```

adding an inner loop of length *n*

Algorithm analysis - nested loop

```
             var M = A[ 0 ];
n:           for ( var i = 0; i < n; ++i ) {
  n            for (var j = 0; j < n; ++j) {
                 if ( A[ i ] >= j * M ) {
                   M = A[ i ];
             }}};
```

Total # instructions (worst-case): $O(n^2)$

Common complexities

- Constant time: $O(1)$
- Logarithmic time: $O(\log n)$
- Linear time: $O(n)$
- Quadratic time: $O(n^2)$
- Also: $O(n \log n)$, $O(n^3)$, etc.

Pipeline and other complexities

- Simple line-oriented UNIX filters: $O(n)$
- Typical sorts: $O(n \log n)$
- Binary search: $O(\log n)$
- Simple sorts: $O(n^2)$
- Brute force "Traveling Salesman": $O(n!)$
- Traveling Salesman (dynamic optimization): $O(2^n)$

see

en.wikipedia.org/wiki/Sorting_algorithm

for much more

Quiz: what is the complexity?

- `grep -ow "Jo" women.txt`
- `grep -ow "Jo" women.txt | wc -l`
- `./split.py women.txt | ./lower.py | sort | uniq -c | sort -rn | head`
- `sort startstation | uniq -c | sort -rn | head`

dionyziz's Rules of Thumb

- Count nesting of loops
- Slower dominates faster, so just count slower
- In general, the worse the big-O, the slower programs will execute with real data
- Improving the big-O of an algorithm often beats using a faster language or faster computer

Revisiting loops: counting steps

a

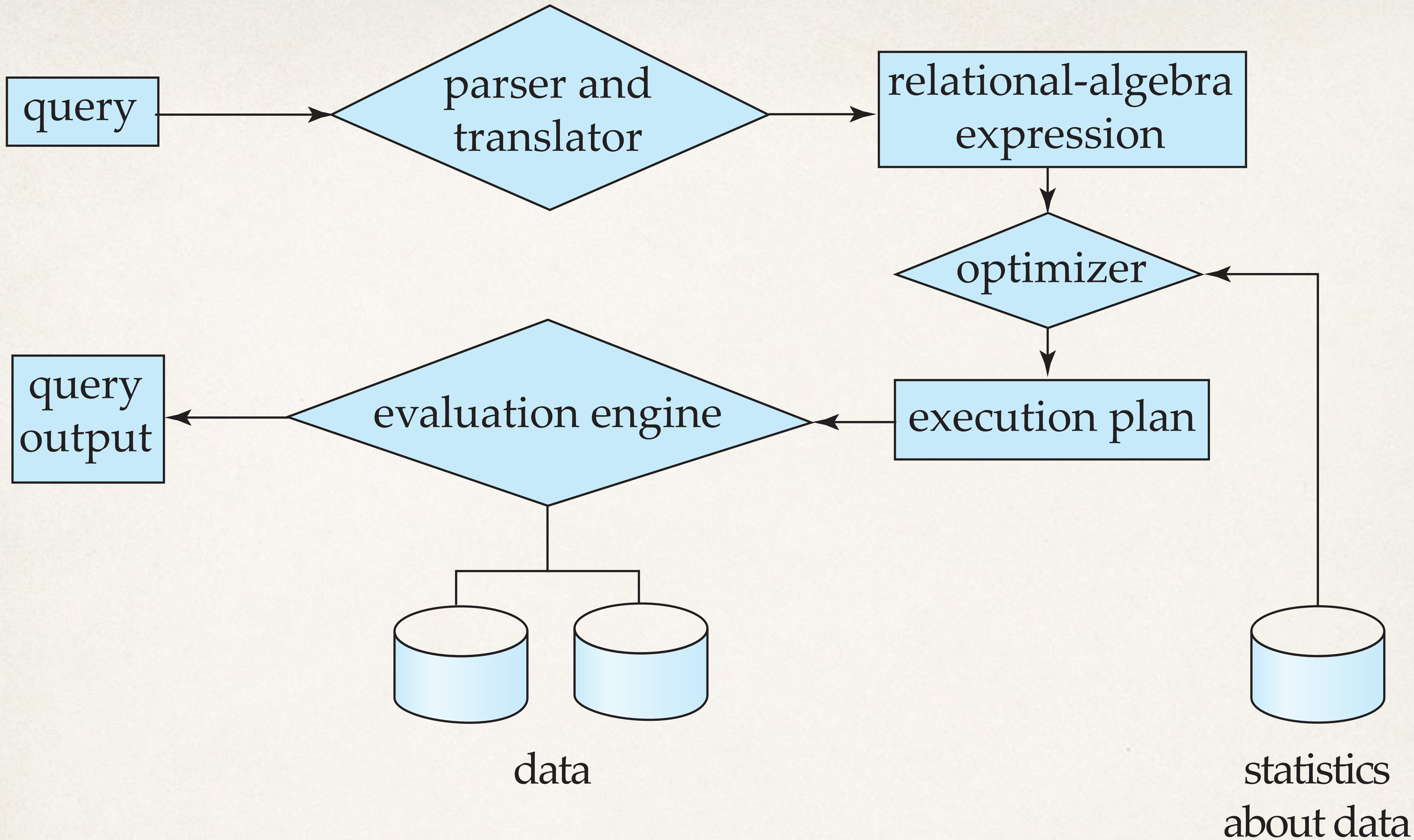
```
for word in line.split():
  stopwords = ['a', 'an', 'the', ...]
  if word in stopwords:
    print(line)
```

b

```
stopwords = ['a', 'an', 'the', ...]
for word in line.split():
  if word in stopwords:
    print(line)
```

RDBMS indexes, query analysis, and optimization

slides borrow liberally from
Chapters 11-13 of Silberschatz et al.



Relational database query processing

image (c) Silbershatz, Kortz, Sudarshan

RDBMS indexing - basics

- Goal is to speed up data access
- **Search key** - attribute set to look up
- **Index file** - (search key, pointer) records
- Index files are smaller than source data
- Index files may be ordered or hashed

RDBMS indexing: metrics

- Access time
- Insertion time
- Deletion time
- Space overhead

Default row-oriented storage

10101	Srinivasan	Comp. Sci.	65000	
12121	Wu	Finance	90000	
15151	Mozart	Music	40000	
22222	Einstein	Physics	95000	
32343	El Said	History	60000	
33456	Gold	Physics	87000	
45565	Katz	Comp. Sci.	75000	
58583	Califieri	History	62000	
76543	Singh	Finance	80000	
76766	Crick	Biology	72000	
83821	Brandt	Comp. Sci.	92000	
98345	Kim	Elec. Eng.	80000	

Dense and sparse indexes

Biology	
Comp. Sci.	
Elec. Eng.	
Finance	
History	
Music	
Physics	

76766	Crick	Biology	72000	
10101	Srinivasan	Comp. Sci.	65000	
45565	Katz	Comp. Sci.	75000	
83821	Brandt	Comp. Sci.	92000	
98345	Kim	Elec. Eng.	80000	
12121	Wu	Finance	90000	
76543	Singh	Finance	80000	
32343	El Said	History	60000	
58583	Califieri	History	62000	
15151	Mozart	Music	40000	
22222	Einstein	Physics	95000	
33465	Gold	Physics	87000	

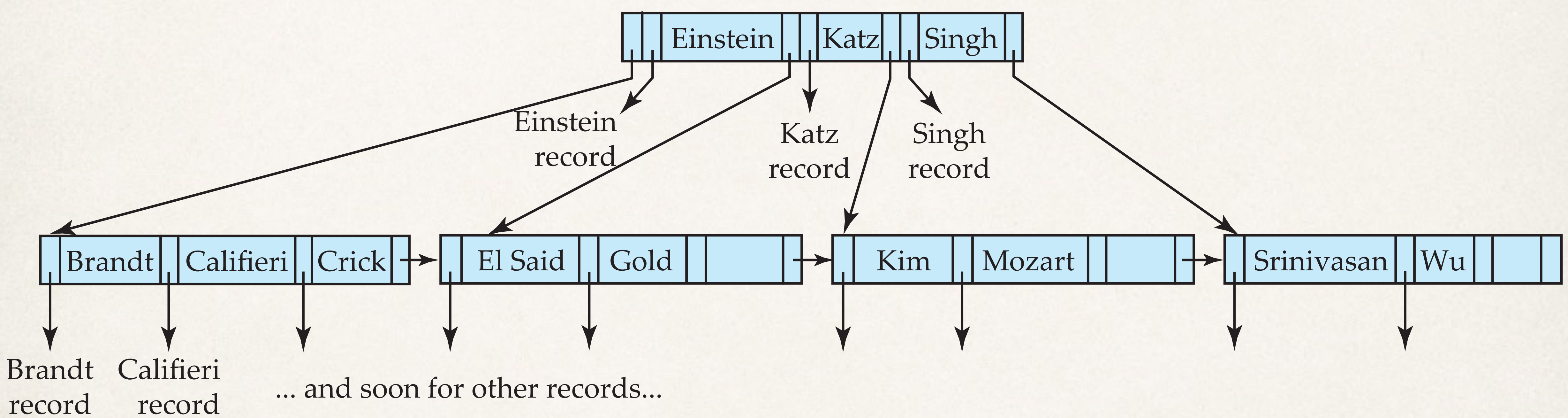
Dense: every department is present in index

10101	
32343	
76766	

10101	Srinivasan	Comp. Sci.	65000	
12121	Wu	Finance	90000	
15151	Mozart	Music	40000	
22222	Einstein	Physics	95000	
32343	El Said	History	60000	
33456	Gold	Physics	87000	
45565	Katz	Comp. Sci.	75000	
58583	Califieri	History	62000	
76543	Singh	Finance	80000	
76766	Crick	Biology	72000	
83821	Brandt	Comp. Sci.	92000	
98345	Kim	Elec. Eng.	80000	

Sparse: only some ids are present in index

B-Tree indexes



Bitmap indexes

record number	*ID*	*gender*	*income_level*
0	76766	m	L1
1	22222	f	L2
2	12121	f	L1
3	15151	m	L4
4	58583	f	L3

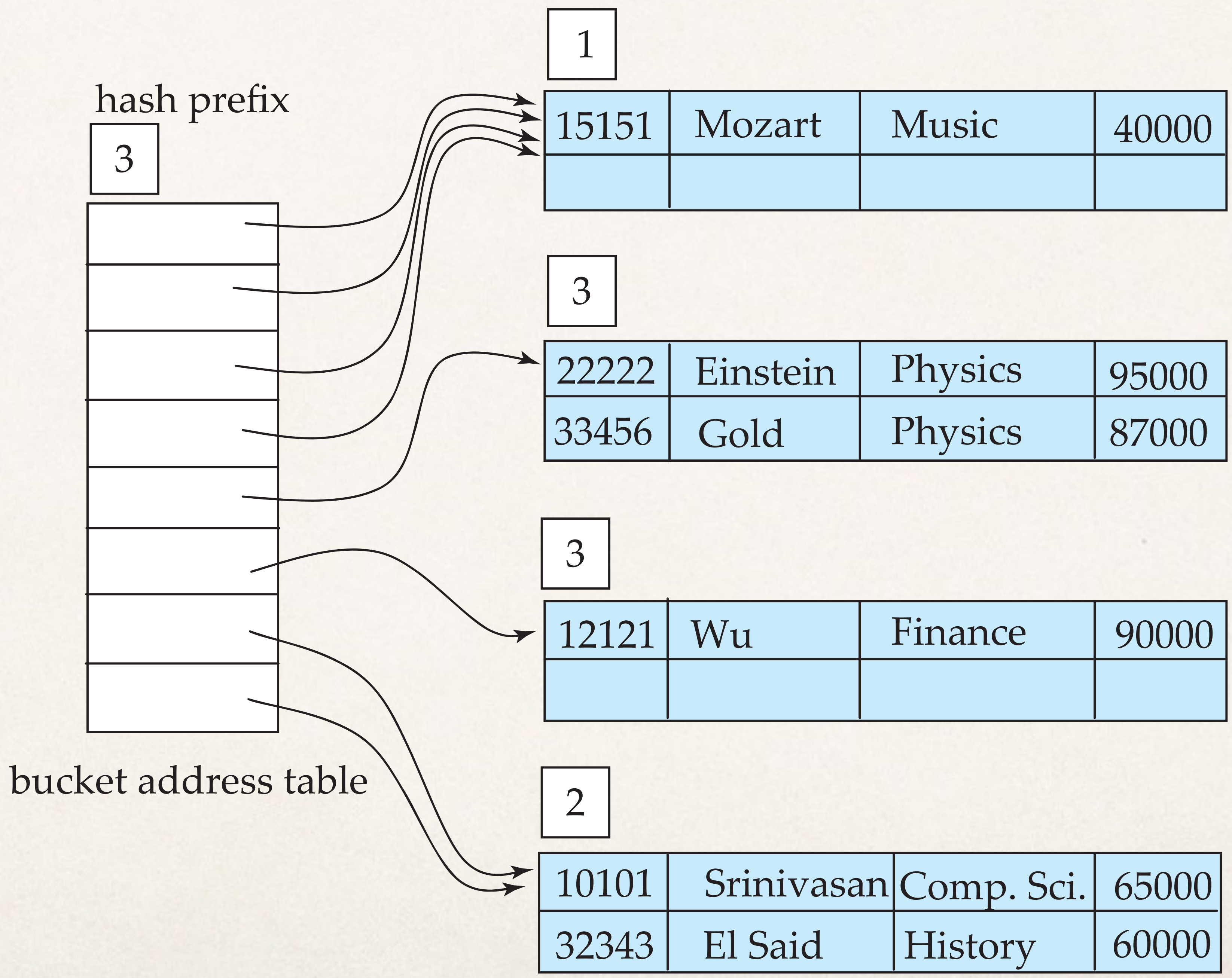
Bitmaps for *gender*

m	10010
f	01101

Bitmaps for *income_level*

L1	10100
L2	01000
L3	00001
L4	00010
L5	00000

Hash indexes



Indexes in practice

- Many variations on these and many more
- Each has different profile on metrics: access time, insertion time, deletion time, space overhead
- Defaults tend to be smart, but at large volumes, indexes require careful consideration
- Easy to create in SQL (DDL)

SQL: CREATE INDEX

```
CREATE INDEX idx_start_station
 ON bikeshare_trips (start_station);

CREATE UNIQUE INDEX idx_student_id
 ON students (id);

CREATE INDEX idx_stations
 ON bikeshare_trips
 (start_station, end_station);
```

Index notes

- Primary and foreign keys are often indexed
- Primary keys should always be UNIQUE
- With large data loads (ETL), run CREATE INDEX after
- Use EXPLAIN SELECT to see db's query plan

Query analysis: basics

- All SQL queries broken down into relational algebra expressions
- Equivalent expressions are generated
- Costs of each are evaluated and optimal plan invoked

Example: instructor search

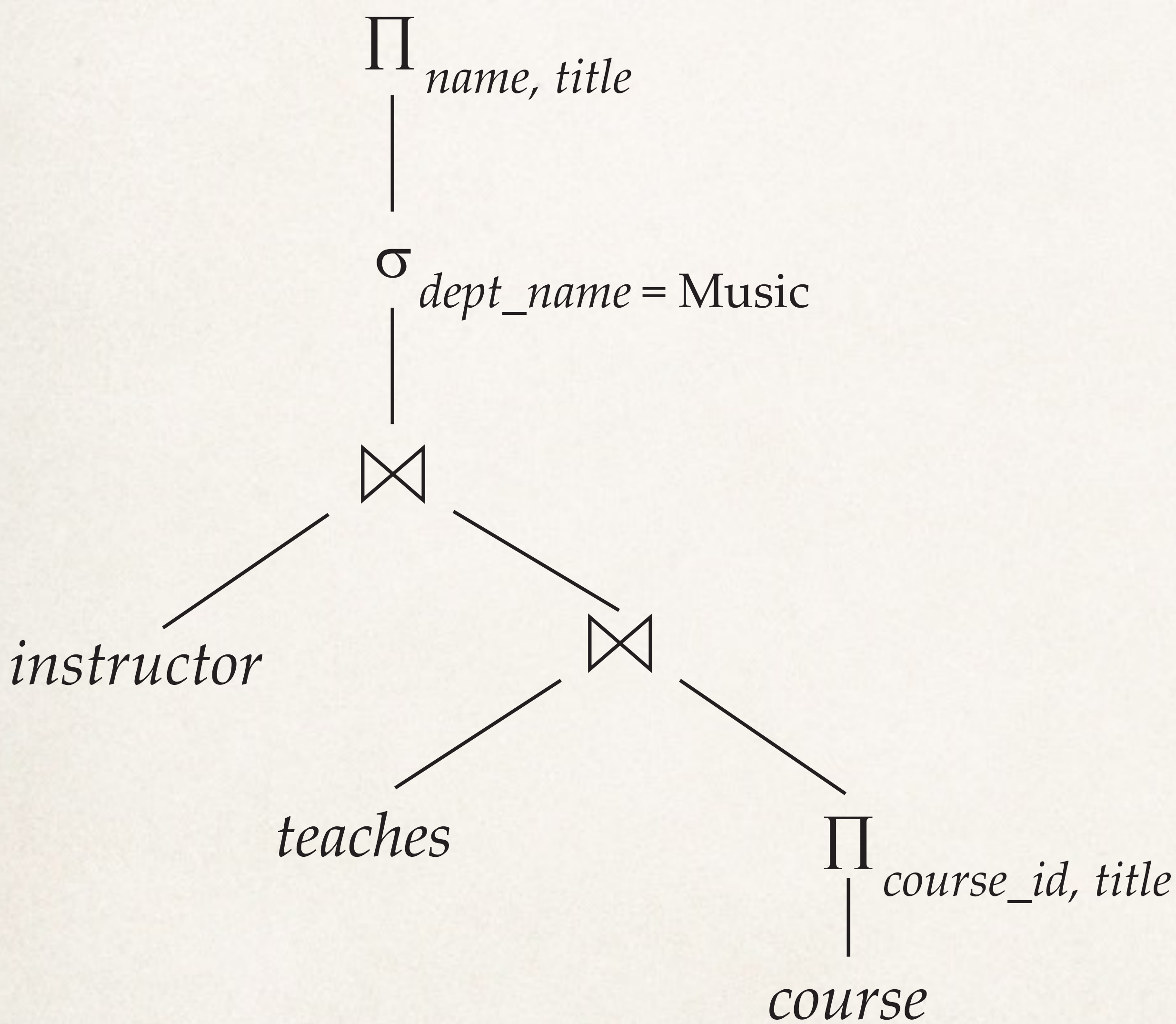
π_{salary}

$\sigma_{salary < 75000;\ \text{use index 1}}$

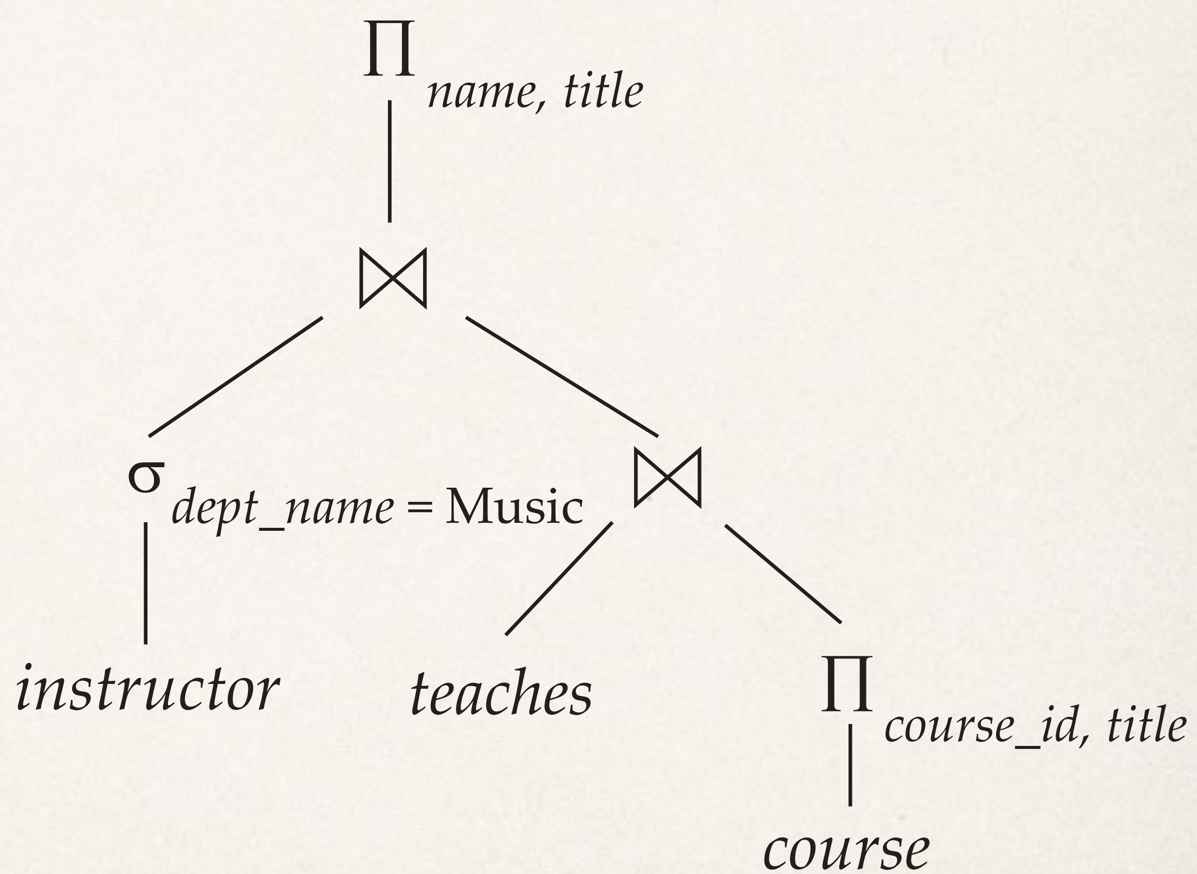
instructor

```
SELECT salary FROM instructor
WHERE salary < 75000;
```

Example: equivalencies



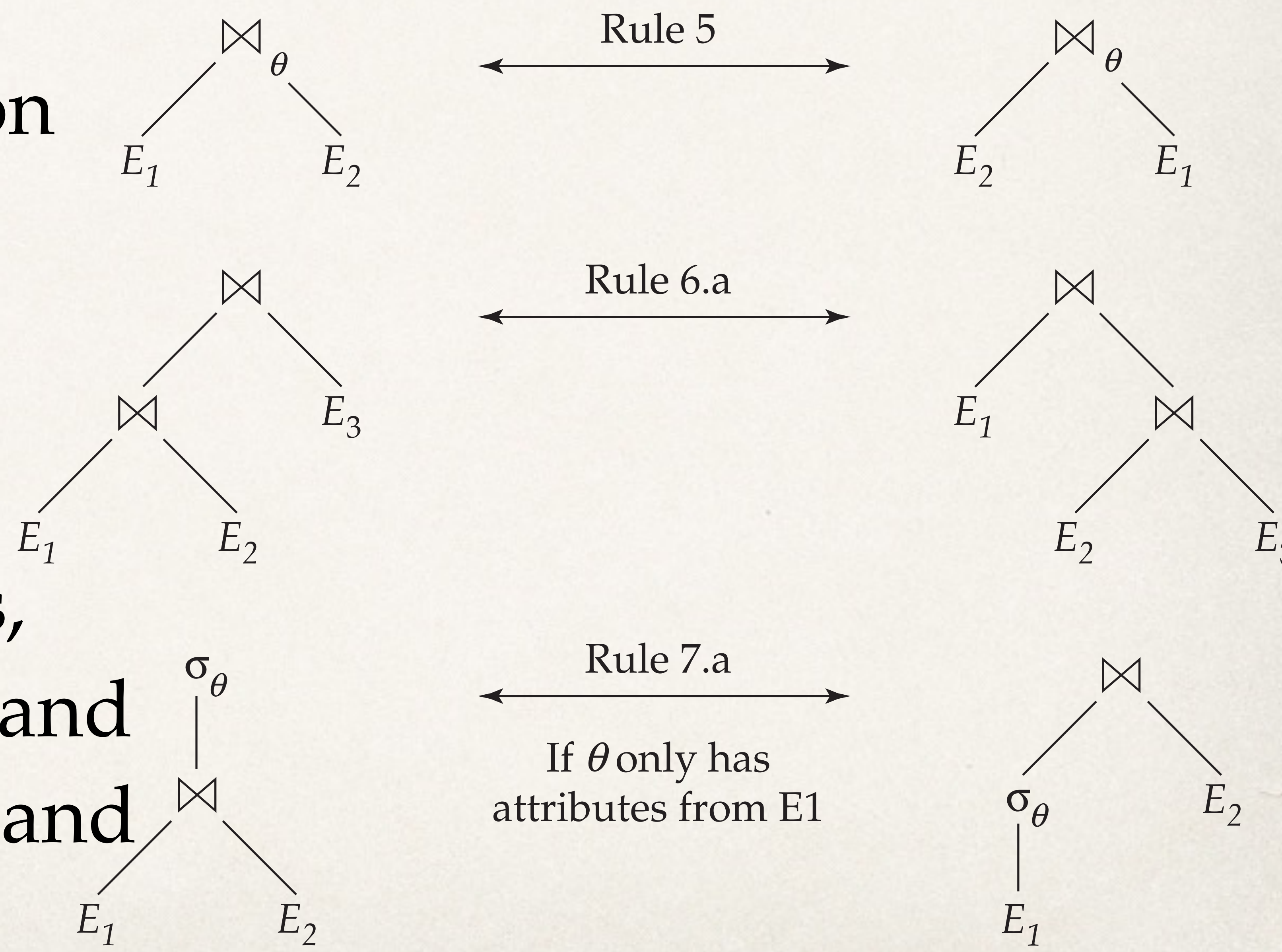
(a) Initial expression tree



(b) Transformed expression tree

Equivalent expressions

- Generate same set of tuples in result relation
- Order irrelevant
- Deconstructing and commuting selections, combining selections and products, associating and commuting joins, etc.



How to compare equivalencies?

- **Cost** of operations is calculated for each step in each plan
- Cost based on disk IO, CPU, size of data
- **File scan** vs. **Index scan**
- Materialization and pipelining

How to evaluate equivalent plans?

- Dynamic optimization
- Recursive
- Shortest weighted path algorithms
- Chapter 9 of Rardin, 2nd ed.
- Ask Goran and Luis for more :)

(remember) Databases provide:

- statistics about stored data
- optimized query processing using those statistics

go to notebook

Basic ETL in SQL

Extract, transform, load

- Formal data wrangling
- **Extract** data from source systems (DBMS, files, CSV, web, etc.)
- **Transform** data according to some consistent structure, schema, or rules
- **Load** data into a target environment (DBMS, CSV, warehouse, Hadoop data lake, etc.)

ETL tools

- Every environment has its own tools and techniques
- Many commercial products support ETL
- All the tools you've been learning do too
- In production, key is to be consistent, repeatable, reproducible in your ETL work

Aspects of ETL - normalizing

- Normalizing de-normalized data
  - Defining normalized schema
  - Extracting unique values from data
  - Transforming existing values to match new rules and formats
  - Transforming existing records to match new value dictionaries
  - Lumping and splitting by time period, category, etc.

Aspects of ETL - de-normalizing

- De-normalizing normalized data
  - Defining de-normalized schema
  - Extracting features for analysis from data
  - Transforming existing values to denormalized patterns
  - Transforming existing records to match new value dictionaries
  - Lumping and splitting by time period, category, etc.

SQL GROUPING SETS, CUBE, ROLLUP

SQL functions for OLAP

- OLAP - Online Analytical Processing
- Standardized in SQL:1999
- Available in Oracle, MSSQL, PostgreSQL 9.5 (recent!)

SQL GROUPING SETS

- SQL functions particularly useful in analysis, ETL, and warehouse development
- constructs GROUP BY-like aggregates for each set specified
  - GROUP BY GROUPING SETS ((e1), (e2), ()) will aggregate all values of e1, then all for e2, then overall
- adds summary counts for all subsets
- enables simple summaries with aggregate functions

SQL ROLLUP and CUBE

- ROLLUP (e1, e2, e3) generates:
  - GROUPING SETS ((e1, e2, e3), (e1, e2), (e1), ())
- CUBE (e1, e2, e3) generates:
  - GROUPING SETS ((e1, e2, e3), (e1, e2), (e1, e3), (e1), (e2, e3), (e2), (e3), ())
- Some OLAP environments optimize for these queries

go to notebook

Exercise 04