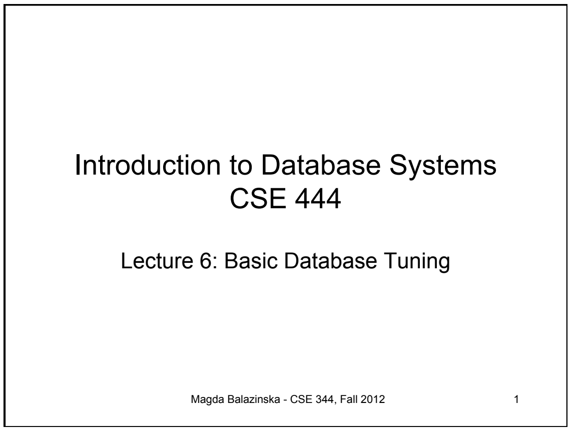
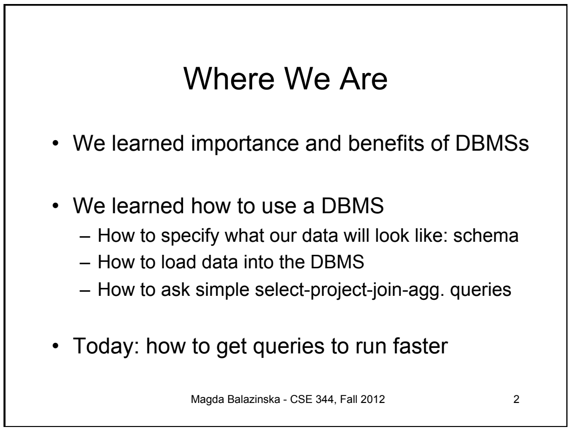


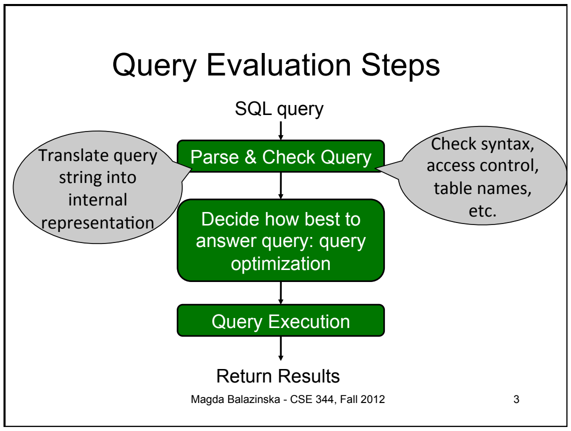
Introduction to Database Systems CSE 444

Lecture 6: Basic Database Tuning



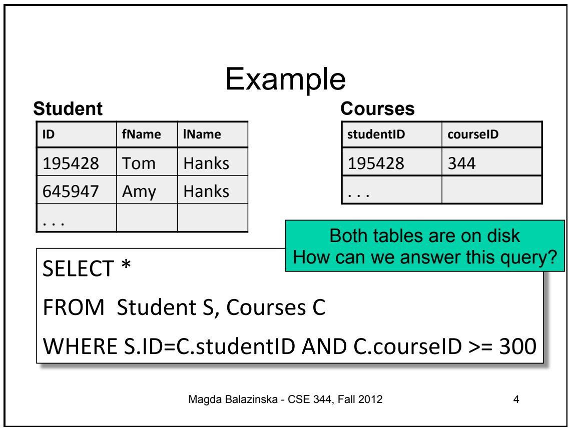
Where We Are

- We learned importance and benefits of DBMSs
- We learned how to use a DBMS
  - How to specify what our data will look like: schema
  - How to load data into the DBMS
  - How to ask simple select-project-join-agg. queries
- Today: how to get queries to run faster



Query Evaluation Steps

SQL query
Parse & Check Query
Decide how best to answer query: query optimization
Query Execution
Return Results
Translate query string into internal representation
Check syntax, access control, table names, etc.



Example

Student

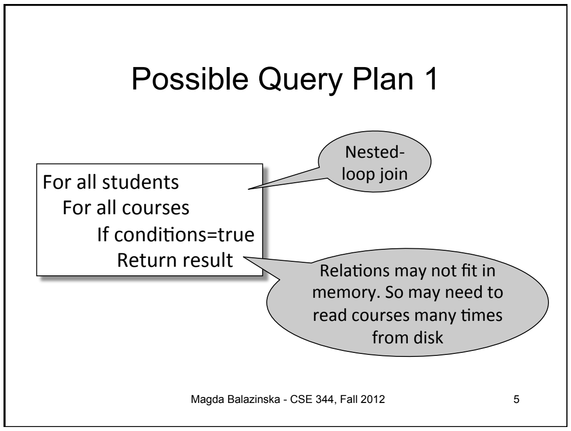
ID	fName	lName
195428	Tom	Hanks
645947	Amy	Hanks
. . .		

Courses

studentID	courseID
195428	344
. . .	

Both tables are on disk
How can we answer this query?

```
SELECT *
FROM  Student S, Courses C
WHERE S.ID=C.studentID AND C.courseID >= 300
```

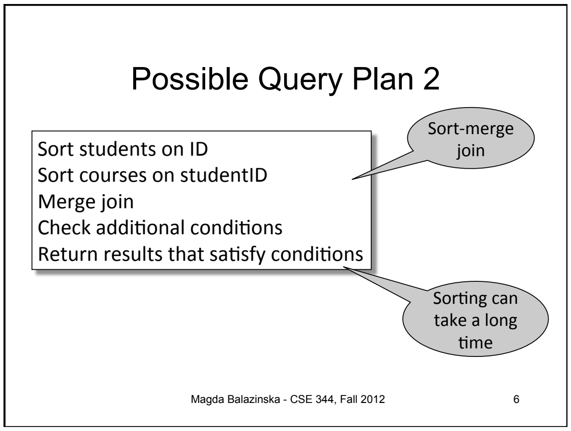


Possible Query Plan 1

```
For all students
   For all courses
        If conditions=true
           Return result
```

Nested-loop join

Relations may not fit in memory. So may need to read courses many times from disk



Possible Query Plan 2

```
Sort students on ID
Sort courses on studentID
Merge join
Check additional conditions
Return results that satisfy conditions
```

Sort-merge join

Sorting can take a long time

Possible Query Plan 3

Create a hash-table of students on ID
Read courses and probe hash table
If match found, check additional conditions
Return results that satisfy the conditions

Hash-join

Still have to read entire relations from disk!

Hash table may not fit in memory

Possible Query Plan 4

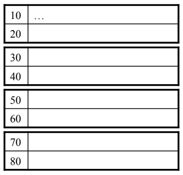
Find and only read from disk courses with courseID >= 300
For each such course, find matching students
Return results

Can we do this?

Yes! But we need **indexes**

Data Storage

- DBMSs store data in **files**
- Most common organization is row-wise storage
- On disk, a file is split into blocks
- Each block contains a set of tuples



In the example, we have 4 blocks with 2 tuples each

Database File Types

The data file can be one of:

- Heap file
  - Unsorted
- Sequential file
  - Sorted according to some attribute(s) called ***key***

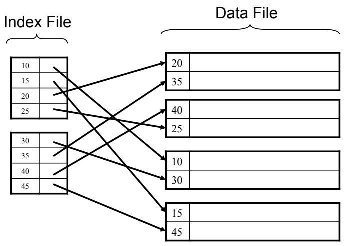
"key" here means something else than "primary key"
Example: ID is primary key for students
But can sort students on last name

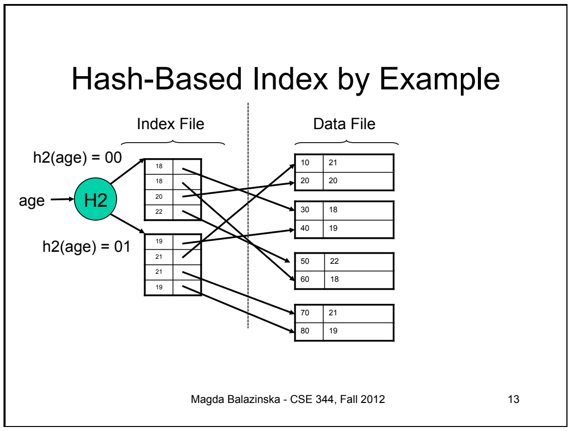
Index

- An **additional** file, that allows fast access to records in the data file given a search key
- The index contains (key, value) pairs:
  - The key = an attribute value (e.g., student ID or name)
  - The value = a pointer to the record

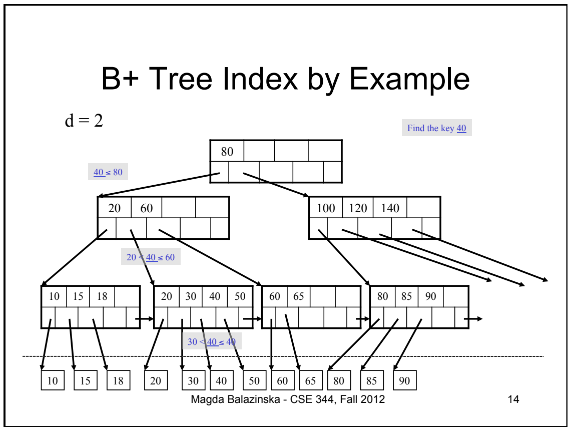
"key" = "search key"

Example of Index

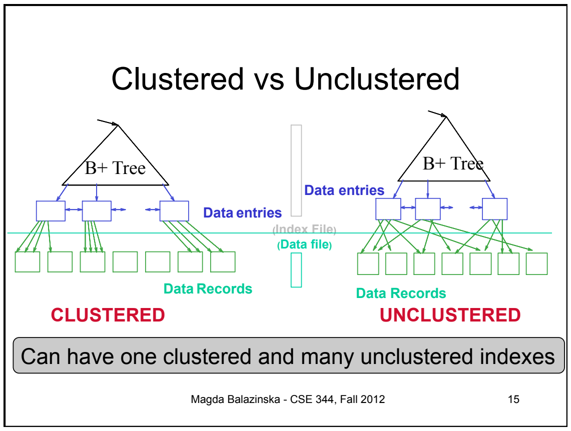




Hash-Based Index by Example

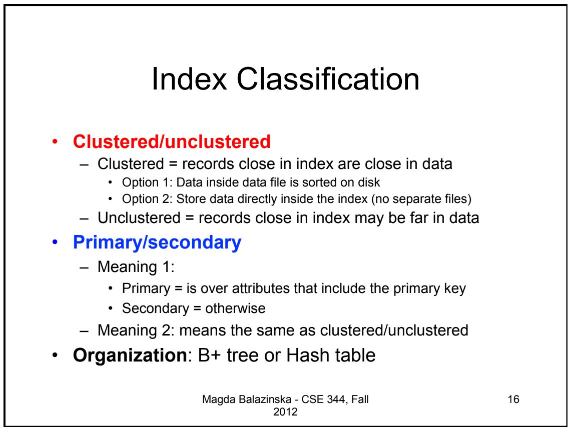


B+ Tree Index by Example



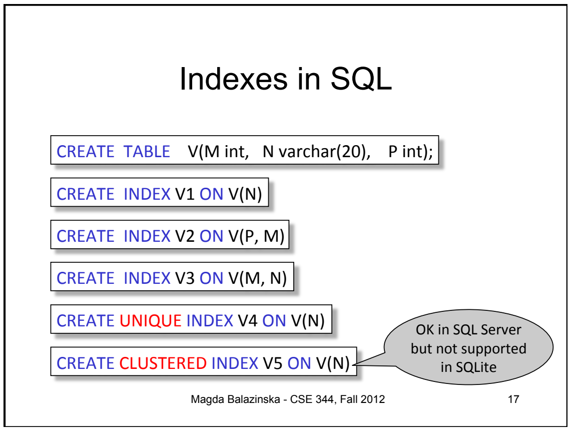
Clustered vs Unclustered

Can have one clustered and many unclustered indexes



Index Classification

- **Clustered/unclustered**
  - Clustered = records close in index are close in data
    - Option 1: Data inside data file is sorted on disk
    - Option 2: Store data directly inside the index (no separate files)
  - Unclustered = records close in index may be far in data
- **Primary/secondary**
  - Meaning 1:
    - Primary = is over attributes that include the primary key
    - Secondary = otherwise
  - Meaning 2: means the same as clustered/unclustered
- **Organization**: B+ tree or Hash table



Indexes in SQL

```
CREATE TABLE   V(M int,   N varchar(20),    P int);
```

```
CREATE  INDEX V1 ON V(N)
```

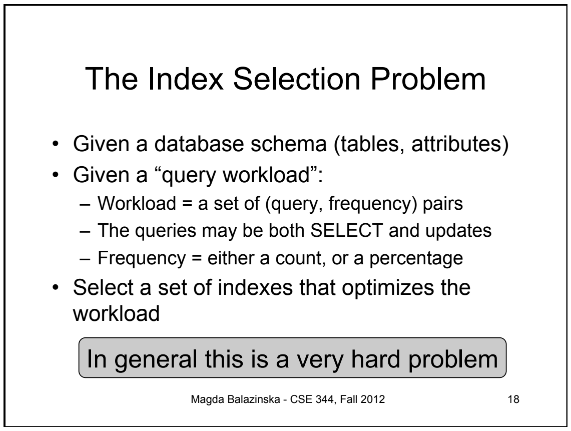
```
CREATE  INDEX V2 ON V(P, M)
```

```
CREATE  INDEX V3 ON V(M, N)
```

```
CREATE UNIQUE INDEX V4 ON V(N)
```

```
CREATE CLUSTERED INDEX V5 ON V(N)
```

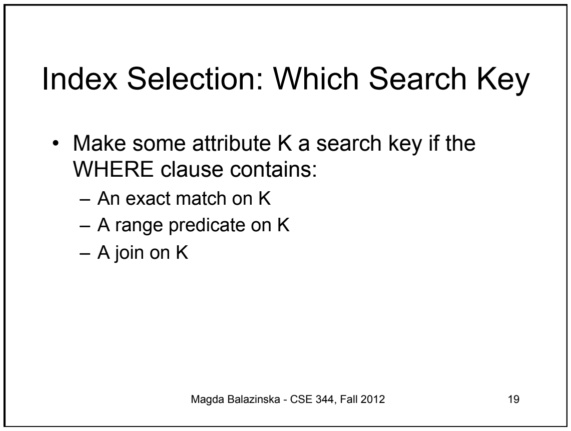
OK in SQL Server but not supported in SQLite



The Index Selection Problem

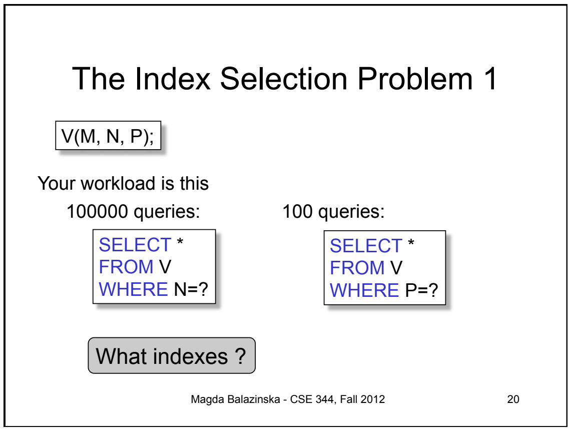
- Given a database schema (tables, attributes)
- Given a "query workload":
  - Workload = a set of (query, frequency) pairs
  - The queries may be both SELECT and updates
  - Frequency = either a count, or a percentage
- Select a set of indexes that optimizes the workload

In general this is a very hard problem



Index Selection: Which Search Key

- Make some attribute K a search key if the WHERE clause contains:
  - An exact match on K
  - A range predicate on K
  - A join on K



The Index Selection Problem 1

V(M, N, P);

Your workload is this

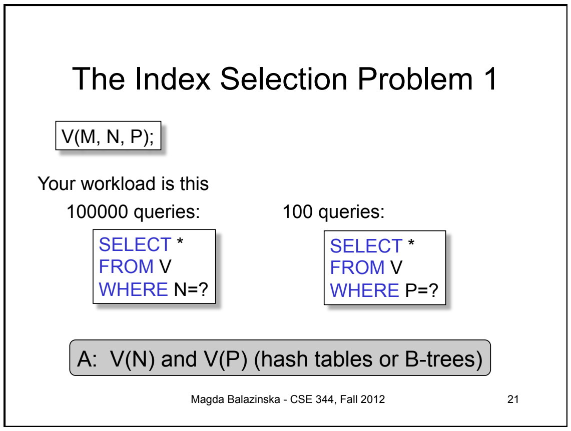
100000 queries:

```
SELECT *
FROM V
WHERE N=?
```

100 queries:

```
SELECT *
FROM V
WHERE P=?
```

What indexes ?



The Index Selection Problem 1

V(M, N, P);

Your workload is this

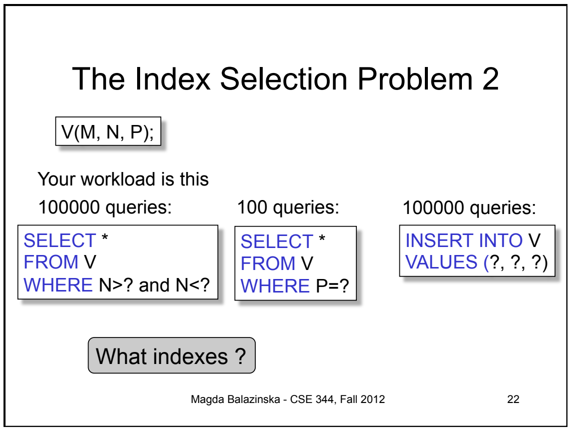
100000 queries:

```
SELECT *
FROM V
WHERE N=?
```

100 queries:

```
SELECT *
FROM V
WHERE P=?
```

A: V(N) and V(P) (hash tables or B-trees)



The Index Selection Problem 2

V(M, N, P);

Your workload is this

100000 queries:

```
SELECT *
FROM V
WHERE N>? and N<?
```

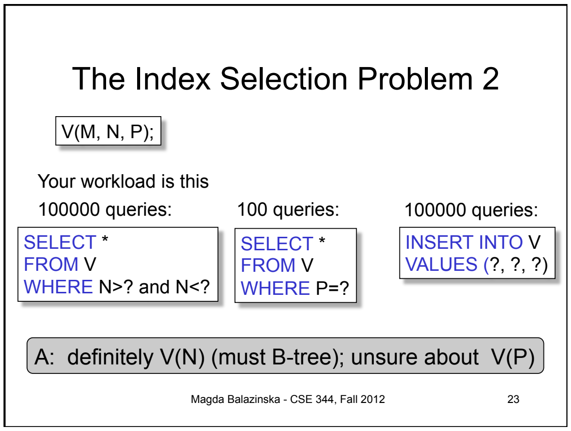
100 queries:

```
SELECT *
FROM V
WHERE P=?
```

100000 queries:

```
INSERT INTO V
VALUES (?, ?, ?)
```

What indexes ?



The Index Selection Problem 2

V(M, N, P);

Your workload is this

100000 queries:

```
SELECT *
FROM V
WHERE N>? and N<?
```

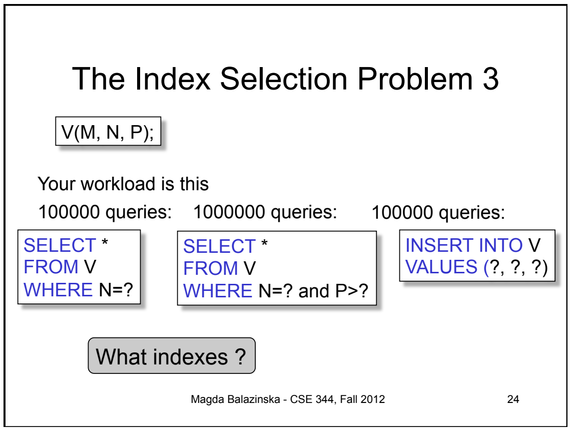
100 queries:

```
SELECT *
FROM V
WHERE P=?
```

100000 queries:

```
INSERT INTO V
VALUES (?, ?, ?)
```

A: definitely V(N) (must B-tree); unsure about V(P)



The Index Selection Problem 3

V(M, N, P);

Your workload is this

100000 queries:

```
SELECT *
FROM V
WHERE N=?
```

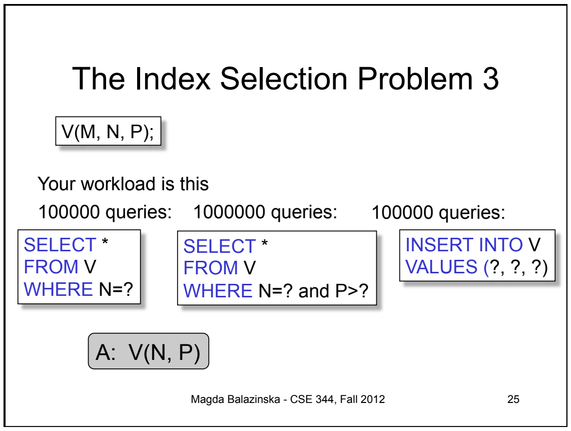
1000000 queries:

```
SELECT *
FROM V
WHERE N=? and P>?
```

100000 queries:

```
INSERT INTO V
VALUES (?, ?, ?)
```

What indexes ?



The Index Selection Problem 3

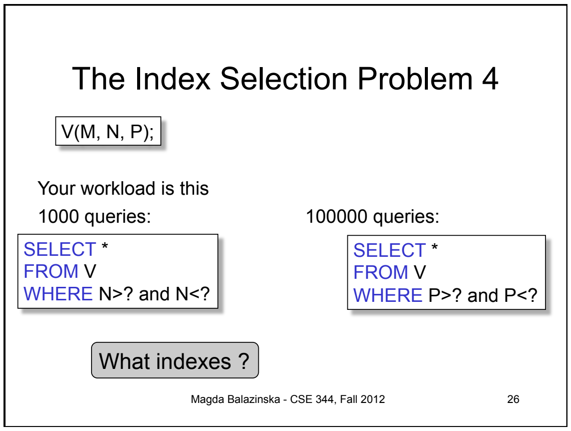
V(M, N, P);

Your workload is this

100000 queries:	1000000 queries:	100000 queries:

SELECT *
FROM V
WHERE N=?
	SELECT *
	FROM V
	WHERE N=? and P>?
		INSERT INTO V
		VALUES (?, ?, ?)

A: V(N, P)



The Index Selection Problem 4

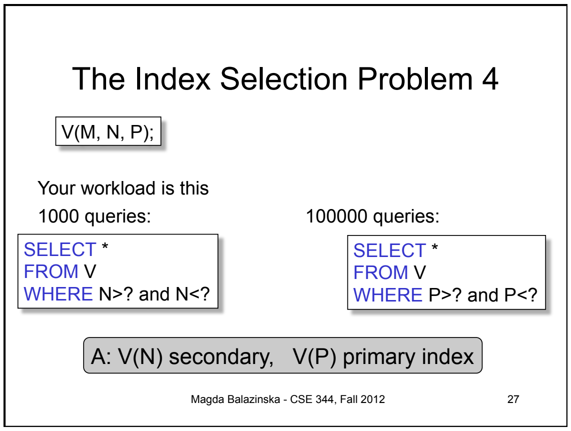
V(M, N, P);

Your workload is this

1000 queries:	100000 queries:

SELECT *
FROM V
WHERE N>? and N<?
	SELECT *
	FROM V
	WHERE P>? and P<?

What indexes ?



The Index Selection Problem 4

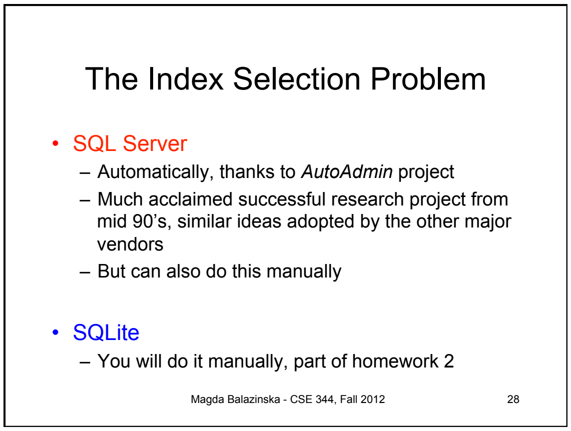
V(M, N, P);

Your workload is this

1000 queries:	100000 queries:

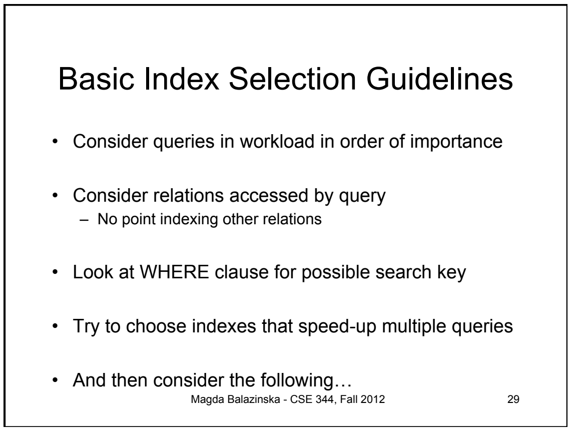
SELECT *
FROM V
WHERE N>? and N<?
	SELECT *
	FROM V
	WHERE P>? and P<?

A: V(N) secondary, V(P) primary index



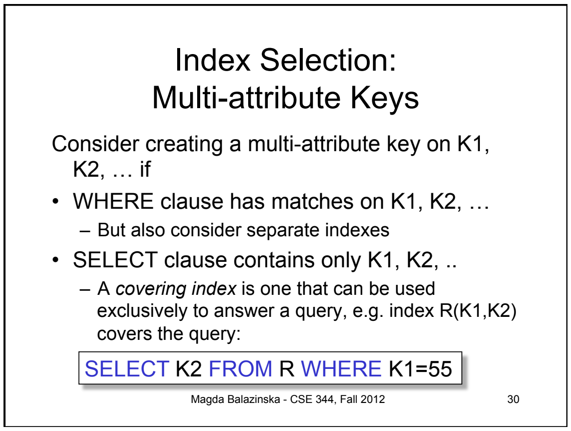
The Index Selection Problem

- SQL Server
  - Automatically, thanks to *AutoAdmin* project
  - Much acclaimed successful research project from mid 90's, similar ideas adopted by the other major vendors
  - But can also do this manually
- SQLite
  - You will do it manually, part of homework 2



Basic Index Selection Guidelines

- Consider queries in workload in order of importance
- Consider relations accessed by query
  - No point indexing other relations
- Look at WHERE clause for possible search key
- Try to choose indexes that speed-up multiple queries
- And then consider the following…

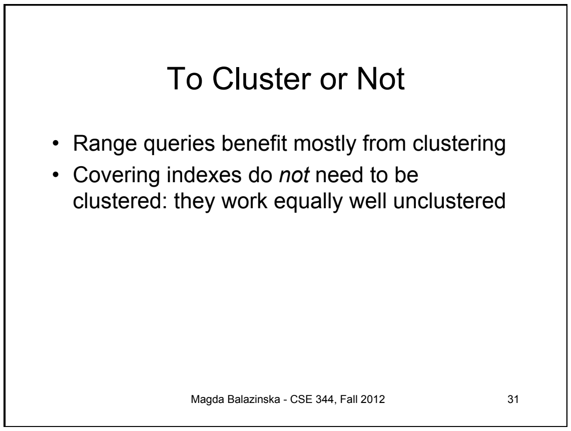


Index Selection: Multi-attribute Keys

Consider creating a multi-attribute key on K1, K2, … if

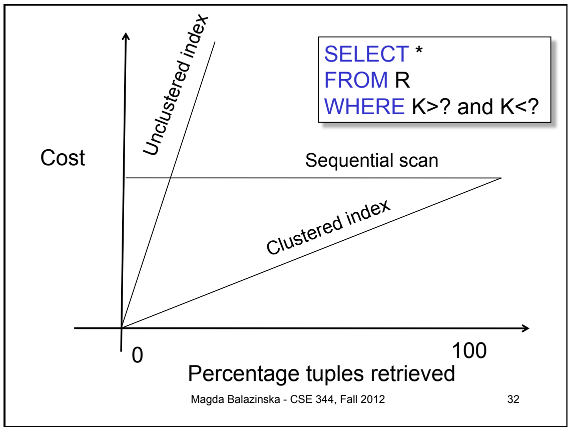
- WHERE clause has matches on K1, K2, …
  - But also consider separate indexes
- SELECT clause contains only K1, K2, ..
  - A *covering index* is one that can be used exclusively to answer a query, e.g. index R(K1,K2) covers the query:

SELECT K2 FROM R WHERE K1=55

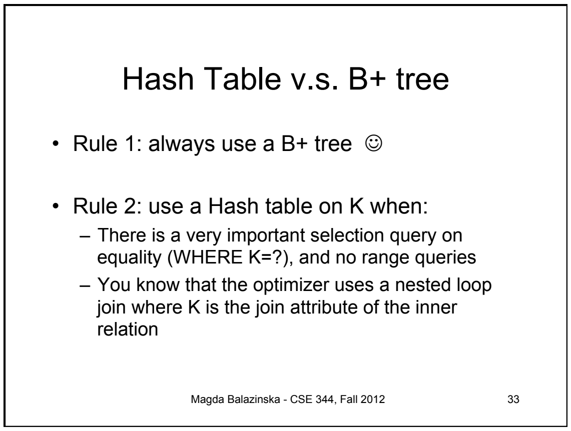


To Cluster or Not

- Range queries benefit mostly from clustering
- Covering indexes do *not* need to be clustered: they work equally well unclustered

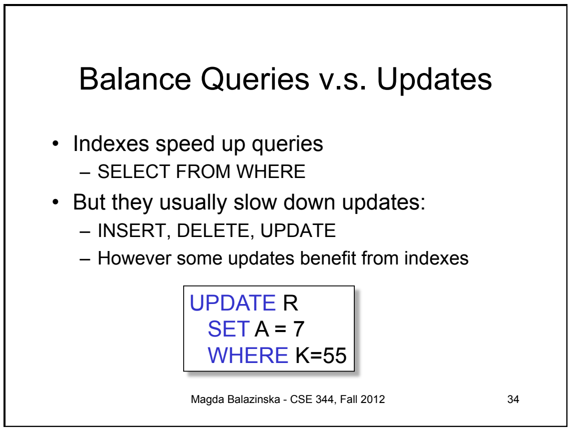


```
SELECT *
FROM R
WHERE K>? and K<?
```



Hash Table v.s. B+ tree

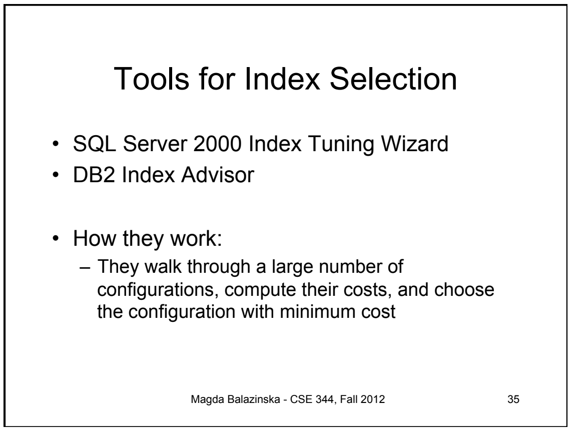
- Rule 1: always use a B+ tree ☺
- Rule 2: use a Hash table on K when:
  - There is a very important selection query on equality (WHERE K=?), and no range queries
  - You know that the optimizer uses a nested loop join where K is the join attribute of the inner relation



Balance Queries v.s. Updates

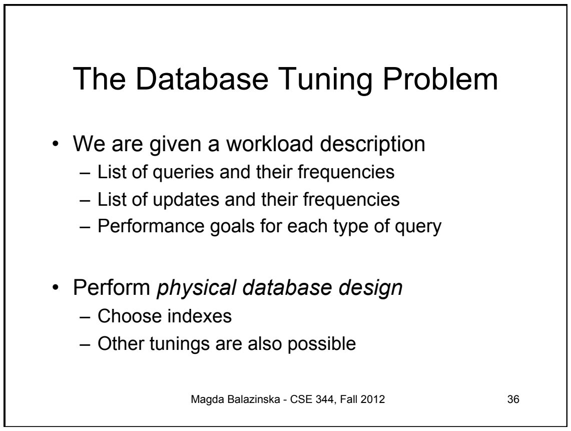
- Indexes speed up queries
  - SELECT FROM WHERE
- But they usually slow down updates:
  - INSERT, DELETE, UPDATE
  - However some updates benefit from indexes

```
UPDATE R
  SET A = 7
  WHERE K=55
```



Tools for Index Selection

- SQL Server 2000 Index Tuning Wizard
- DB2 Index Advisor
- How they work:
  - They walk through a large number of configurations, compute their costs, and choose the configuration with minimum cost



The Database Tuning Problem

- We are given a workload description
  - List of queries and their frequencies
  - List of updates and their frequencies
  - Performance goals for each type of query
- Perform *physical database design*
  - Choose indexes
  - Other tunings are also possible