CHALMERS
UNIVERSITY OF TECHNOLOGY

UNIVERSITY OF GOTHENBURG

Views, Indices and Transactions

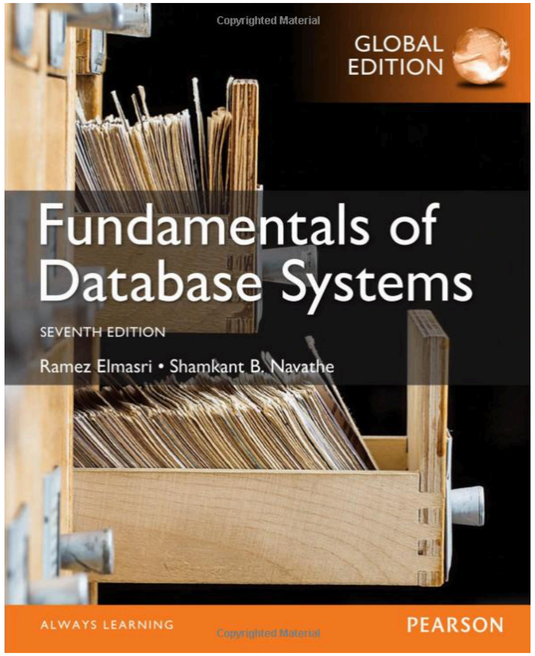
LECTURE 10

Dr. Philipp Leitner

philipp.leitner@chalmers.se

@xLeitix

LECTURE 10



Covers …

- Transactions (Chapter 20)
- Views (parts of Chapter 7)
- Indices (theory covered in Chapter 17, we focus more on how to use them)

What we will be covering

How to avoid multiple concurrent users interfering

Transactions

How to deal with performance issues in databases

Views and Indices

Multi-User Databases

Single-user DBMS

At most one user at a time can use the system

So far we have used our DBMS in this way

Multi-user DBMS

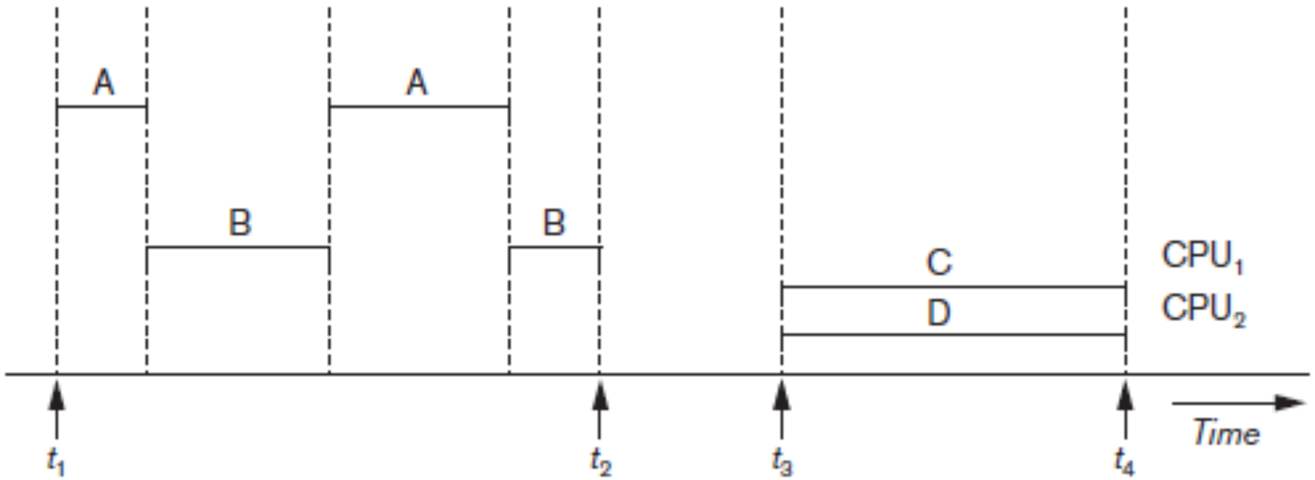
Many users can access the DBMS concurrently

So far we did not think about the potential of other users running competing updates in parallel

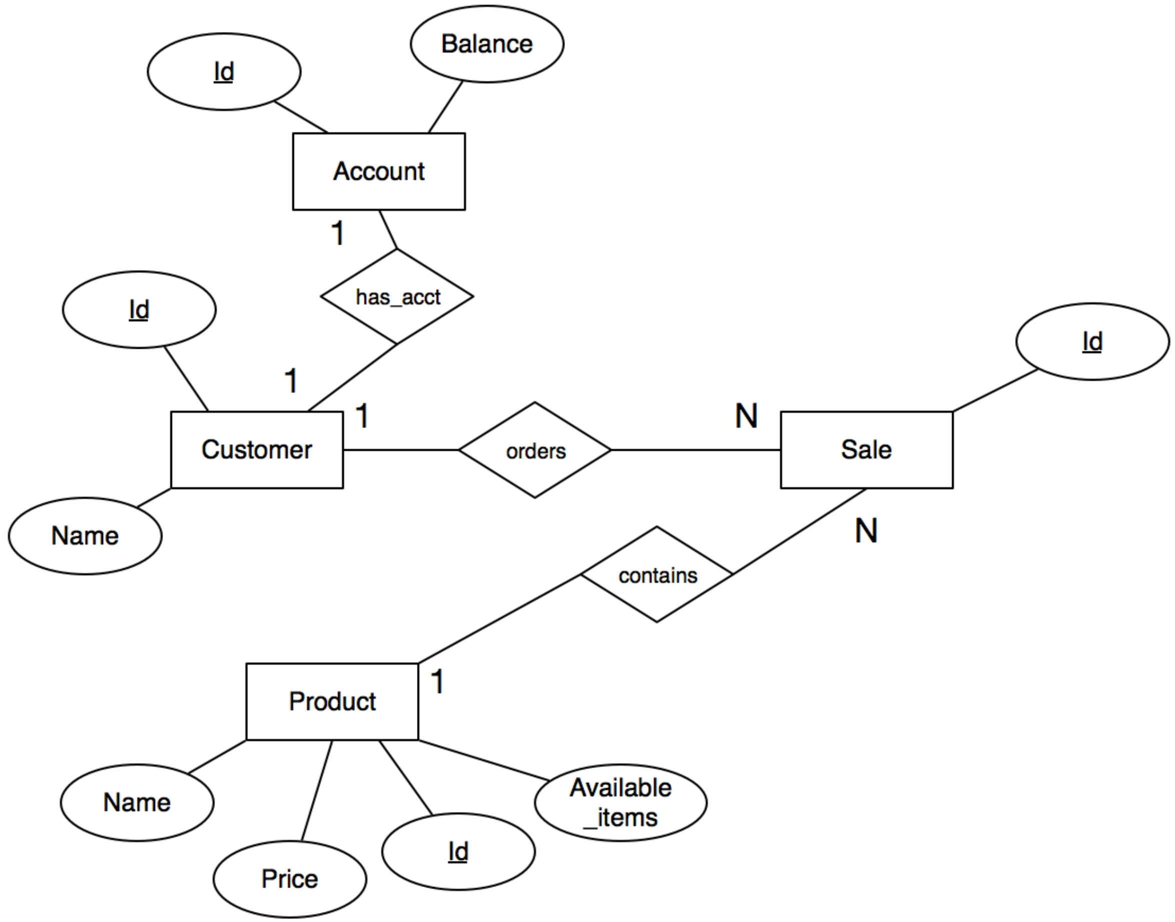
Concurrency in DBMSs

DBMSs are typically designed to support **multiple concurrent users**

Transactions are way to ensure consistency of interleaved processes



Example


Id
Balance
Account
1
has_acct
Id
1
1
Customer
orders
N
Sale
Id
Name
N
contains
Product
1
Name
Price
Id
Available
_items

Transactions

Transactions are an **atomic set** of statements:

- Either **all statements** should be executed, or **none of them**
- No other statements should be executed **between** statements in a transaction

Transactions Boundaries

Transactions are typically defined through **transaction boundaries**

Three kinds of markings:

BEGIN TRANSACTION

COMMIT TRANSACTION (save changes to database)

ROLLBACK TRANSACTION (discard changes)

Transactions in SQL

BEGIN used to start a **new transaction**

COMMIT used to **save** changes to disk

ROLLBACK used to **undo** changes

ACID Properties of SQL Transactions

Atomicity

All changes are applied, or nothing is applied

Consistency

Database is always in a consistent state (no “in-between” time)

Isolation

Concurrency control - guarantees that concurrent transactions are not interfering

Durability

Once a change is applied, it remains even through failures

Some Transactional Problems: Lost Updates

(a)

T_1	T_2
read_item(X); $X := X - N$;	
	read_item(X); $X := X + M$;
write_item(X); read_item(Y);	
	write_item(X);
$Y := Y + N$; write_item(Y);	

Time

Item X has an incorrect value because its update by T_1 is *lost* (overwritten).

Some Transactional Problems: Temporary Updates

(b)

Time

T_1	T_2

read_item(X);
$X := X - N$;
write_item(X);	

	read_item(X);
	$X := X + M$;
	write_item(X);

read_item(Y);	

Transaction T_1 fails and must change the value of X back to its old value; meanwhile T_2 has read the *temporary* incorrect value of X.

Some Transactional Problems: Incorrect Summaries

(c)

T_1	T_3

	sum := 0;
	read_item(*A*);
	sum := *sum* + *A*;
	⋮

read_item(*X*);
X := *X* − *N*;
write_item(*X*);	

	read_item(*X*);
	sum := *sum* + *X*;
	read_item(*Y*);
	sum := *sum* + *Y*;

read_item(*Y*);
Y := *Y* + *N*;
write_item(*Y*);	

T_3 reads *X* after *N* is subtracted and reads *Y* before *N* is added; a wrong summary is the result (off by *N*).

Some Transactional Problems: Nonrepeatable Reads

Transaction T reads the same item twice
Value is changed by another transaction T′ between the two reads

T receives **different values for the two reads** of the same item

Database Isolation Levels

Dirty read

Read operations can return **uncommited data** of other transactions (temporary updates)

Nonrepeatable reads

Reading the **same row twice** within a transaction may show **different attribute values**

Phantom reads

Executing the **same query twice** within a transaction may lead to a **differing number of results**

Database Isolation Levels

	Type of Violation		
Isolation Level	Dirty Read	Nonrepeatable Read	Phantom
READ UNCOMMITTED	Yes	Yes	Yes
READ COMMITTED	No	Yes	Yes
REPEATABLE READ	No	No	Yes
SERIALIZABLE	No	No	No

In Postgres

```
SET TRANSACTION isolation level <mode>;
```

Where <mode> is one of:

- `SERIALIZABLE`
- `REPEATABLE READ`
- `READ COMMITED` (default)
- `READ UNCOMMITED`

In Postgres

For example:

```
BEGIN;
SET TRANSACTION isolation level SERIALIZABLE;
INSERT INTO …
INSERT INTO …
INSERT INTO …
COMMIT;
```

Short Quiz on Kahoot!

Optimizing Database Performance

In practical DBMS usage, **performance** is a permanent concern

Performance basically comes in two (related) flavors:

Avoiding excessive joins (**views**)
Avoiding full-table scans (**indices**)

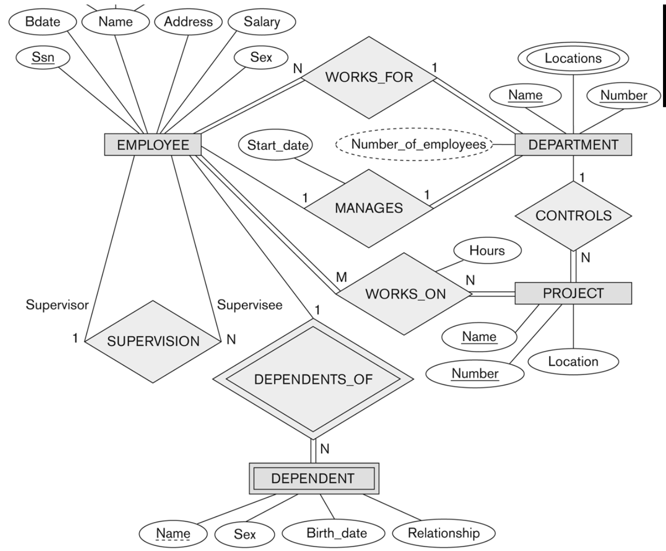


Figure 3.2
An ER schema diagram for the COMPANY database. The diagrammatic notation is introduced gradually throughout this chapter.

Excessive Joining

Assume you are building a web app for project management

On login, you need to:

> **For the currently logged in employee, show links to all department web pages that the employee is not associated with, but which collaborate in a project with this employee**

(may easily be a very slow query, especially if there are many projects)

Excessive Joining

For the currently logged in employee, show links to all department web pages that the employee is not associated with, but which collaborate in a project with this employee

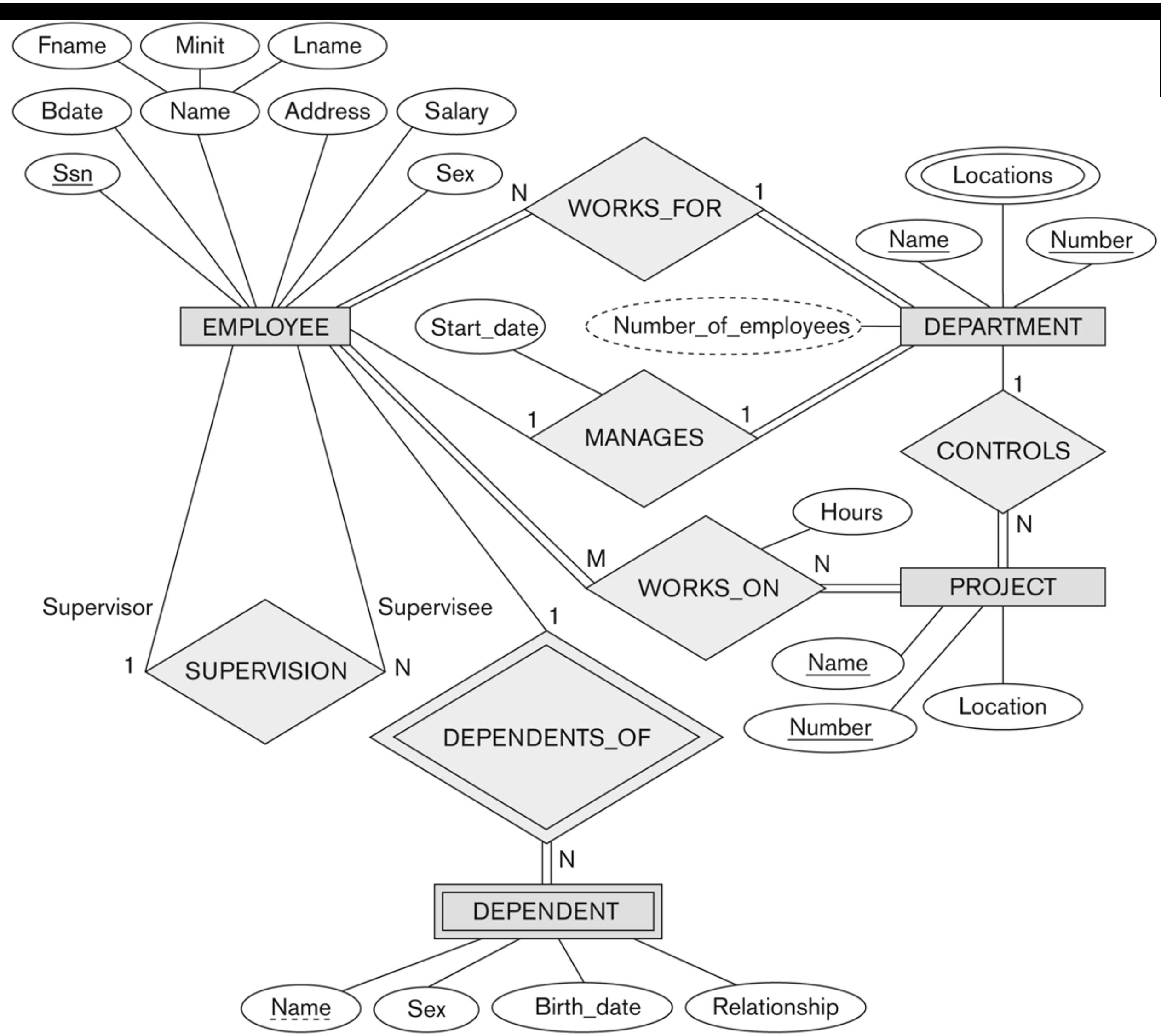


Figure 3.2
An ER schema diagram for the COMPANY database. The diagrammatic notation is introduced gradually throughout this chapter.

Excessive Joining

One problem of nice, **normalized database** schemas:

- In practice queries we often end up joining many, many tables
- And that can be very slow

Possible solutions:

- Caching results
- Controlled redundancy
- **(Materialized) Views**
  - "Virtual" table
  - Basically a query that's used so often that it is "stored" to disk

Examples

CREATE VIEW

```
V1:   CREATE VIEW   WORKS_ON1
      AS SELECT     Fname, Lname, Pname, Hours
         FROM       EMPLOYEE, PROJECT, WORKS_ON
         WHERE      Ssn=Essn AND Pno=Pnumber;

V2:   CREATE VIEW   DEPT_INFO(Dept_name, No_of_emps, Total_sal)
      AS SELECT     Dname, COUNT (*), SUM (Salary)
         FROM       DEPARTMENT, EMPLOYEE
         WHERE      Dnumber=Dno
         GROUP BY   Dname;
```

Views

Such views can for practical purposes be used like any other table **for querying**

Updates are (virtually) unsupported, except in narrow cases:

- View cannot be based on join
- View cannot use `DISTINCT`
- View cannot use aggregation
- (+ a few other minor restrictions)

Materialized views are not considered redundant, because the DB manages them!

Implementation of Views

Two implementation techniques for views:

Query modification

View is basically a saved query, query gets rewritten and executed for each request

No performance gain, just usability

Materialized view

View gets stored to database as a special kind of table

Needs more space, but faster

View Materialization

Different ways to handle materialization:

- **Immediate update strategy** updates a view as soon as the base tables are changed
  - Slower inserts
- **Lazy update strategy** updates the view when needed by a view query
  - Slower queries
- **Periodic / on demand update strategy** updates the view periodically or on demand
  - Potential for inconsistencies

In PGSQL

To create a non-materialized view:

```
CREATE VIEW <query>;
```

To create a materialized view:

```
CREATE MATERIALIZED VIEW <query>;
```

Refresh a materialized view:

```
REFRESH MATERIALIZED VIEW <viewname>;
```

(uses the on-demand update strategy)

Full Table Scans

Assume the following simple query:

```
SELECT * FROM EMPLOYEE
WHERE Lname = 'Smith';
```

This is actually a **fairly expensive** query for the DMBS

It needs to look at **each employee row** and determine if the Lname matches "Smith"

- For most rows, the result will likely be "no"

We call this a **full table scan**

EMPLOYEE

Fname	Minit	Lname	Ssn	Bdate	Address	Sex	Salary	Super_ssn	Dno
John	B	Smith	123456789	1965-01-09	731 Fondren, Houston, TX	M	30000	333445555	5
Franklin	T	Wong	333445555	1955-12-08	638 Voss, Houston, TX	M	40000	888665555	5
Alicia	J	Zelaya	999887777	1968-01-19	3321 Castle, Spring, TX	F	25000	987654321	4
Jennifer	S	Wallace	987654321	1941-06-20	291 Berry, Bellaire, TX	F	43000	888665555	4
Ramesh	K	Narayan	666884444	1962-09-15	975 Fire Oak, Humble, TX	M	38000	333445555	5
Joyce	A	English	453453453	1972-07-31	5631 Rice, Houston, TX	F	25000	333445555	5
Ahmad	V	Jabbar	987987987	1969-03-29	980 Dallas, Houston, TX	M	25000	987654321	4
James	E	Borg	888665555	1937-11-10	450 Stone, Houston, TX	M	55000	NULL	1

DEPARTMENT

Dname	Dnumber	Mgr_ssn	Mgr_start_date
Research	5	333445555	1988-05-22
Administration	4	987654321	1995-01-01
Headquarters	1	888665555	1981-06-19

DEPT_LOCATIONS

Dnumber	Dlocation
1	Houston
4	Stafford
5	Bellaire
5	Sugarland
5	Houston

EMPLOYEE

Fname	Minit	Lname	Ssn	Bdate	Address	Sex	Salary	Super_ssn	Dno
John	B	Smith	123456789	1965-01-09	731 Fondren, Houston, TX	M	30000	333445555	5
Franklin	T	Wong	333445555	1955-12-08	638 Voss, Houston, TX	M	40000	888665555	5
Alicia	J	Zelaya	999887777	1968-01-19	3321 Castle, Spring, TX	F	25000	987654321	4
Jennifer	S	Wallace	987654321	1941-06-20	291 Berry, Bellaire, TX	F	43000	888665555	4
Ramesh	K	Narayan	666884444	1962-09-15	975 Fire Oak, Humble, TX	M	38000	333445555	5
Joyce	A	English	453453453	1972-07-31	5631 Rice, Houston, TX	F	25000	333445555	5
Ahmad	V	Jabbar	987987987	1969-03-29	980 Dallas, Houston, TX	M	25000	987654321	4
James	E	Borg	888665555	1937-11-10	450 Stone, Houston, TX	M	55000	NULL	1

DEPARTMENT

Dname	Dnumber	Mgr_ssn	Mgr_start_date
Research	5	333445555	1988-05-22
Administration	4	987654321	1995-01-01
Headquarters	1	888665555	1981-06-19

DEPT_LOCATIONS

Dnumber	Dlocation
1	Houston
4	Stafford
5	Bellaire
5	Sugarland
5	Houston

EMPLOYEE

Fname	Minit	Lname	Ssn	Bdate	Address	Sex	Salary	Super_ssn	Dno
John	B	Smith	123456789	1965-01-09	731 Fondren, Houston, TX	M	30000	333445555	5
Franklin	T	Wong	333445555	1955-12-08	638 Voss, Houston, TX	M	40000	888665555	5
Alicia	J	Zelaya	999887777	1968-01-19	3321 Castle, Spring, TX	F	25000	987654321	4
Jennifer	S	Wallace	987654321	1941-06-20	291 Berry, Bellaire, TX	F	43000	888665555	4
Ramesh	K	Narayan	666884444	1962-09-15	975 Fire Oak, Humble, TX	M	38000	333445555	5
Joyce	A	English	453453453	1972-07-31	5631 Rice, Houston, TX	F	25000	333445555	5
Ahmad	V	Jabbar	987987987	1969-03-29	980 Dallas, Houston, TX	M	25000	987654321	4
James	E	Borg	888665555	1937-11-10	450 Stone, Houston, TX	M	55000	NULL	1

DEPARTMENT

Dname	Dnumber	Mgr_ssn	Mgr_start_date
Research	5	333445555	1988-05-22
Administration	4	987654321	1995-01-01
Headquarters	1	888665555	1981-06-19

DEPT_LOCATIONS

Dnumber	Dlocation
1	Houston
4	Stafford
5	Bellaire
5	Sugarland
5	Houston

EMPLOYEE

Fname	Minit	Lname	Ssn	Bdate	Address	Sex	Salary	Super_ssn	Dno
John	B	Smith	123456789	1965-01-09	731 Fondren, Houston, TX	M	30000	333445555	5
Franklin	T	Wong	333445555	1955-12-08	638 Voss, Houston, TX	M	40000	888665555	5
Alicia	J	Zelaya	999887777	1968-01-19	3321 Castle, Spring, TX	F	25000	987654321	4
Jennifer	S	Wallace	987654321	1941-06-20	291 Berry, Bellaire, TX	F	43000	888665555	4
Ramesh	K	Narayan	666884444	1962-09-15	975 Fire Oak, Humble, TX	M	38000	333445555	5
Joyce	A	English	453453453	1972-07-31	5631 Rice, Houston, TX	F	25000	333445555	5
Ahmad	V	Jabbar	987987987	1969-03-29	980 Dallas, Houston, TX	M	25000	987654321	4
James	E	Borg	888665555	1937-11-10	450 Stone, Houston, TX	M	55000	NULL	1

....

('n' lookups necessary where 'n' is the number of rows)

DEPARTMENT

Dname	Dnumber	Mgr_ssn	Mgr_start_date
Research	5	333445555	1988-05-22
Administration	4	987654321	1995-01-01
Headquarters	1	888665555	1981-06-19

DEPT_LOCATIONS

Dnumber	Dlocation
1	Houston
4	Stafford
5	Bellaire
5	Sugarland
5	Houston

Computational Complexity of Full Table Scans

In terms of **computational complexity**:

A full table scan is in the complexity class *O(n)*

Where *n* is the number of rows in the table

Indices

Indices (or **indexes**) solve this problem:

Separate data structure that is **efficiently** able to answer the question *"Which row(s) in this table are equal to X?"*

O(1) complexity!

Indices

Indices (or **indexes**) solve this problem:

- Separate data structure that is **efficiently** able to answer the question *“Which row(s) in this table are equal to X?”*

- *O(1)* complexity!

Implemented through hashing

Implemented through **hashing**

- Basics of hashing: in your algorithms class
- For details of index implementation refer to Chapter 17 in the book

CREATE INDEX

In SQL you may create an index for any attribute or combination of attributes using the `CREATE INDEX` statement

```
CREATE INDEX <IDX_NAME> ON <TABLE>(<ATT_LIST>)
```

```
CREATE INDEX loc_idx ON PROJECT(Location)
```

(creating an index on an already existing, large table may take a while)

CREATE UNIQUE INDEX

There is a different variation for indices on keys

```
CREATE UNIQUE INDEX <IDX_NAME> ON <TABLE>(<ATT_LIST>)
```

```
CREATE UNIQUE INDEX name_idx ON DEPENDENT(name)
```

(note that primary keys are **indexed by default** in virtually all database implementations)

DROP INDEX

An index can also be deleted again using `DROP INDEX` (except for default PK indices, those cannot be dropped)

```
DROP INDEX <IDX_NAME>

DROP INDEX loc_idx
```

Advantages and Disadvantages

Creating an index is a trade-off

By design, querying **on this specific row** avoids full table scans and is much faster, but:

- **There is no gain for other queries**
  - Trivial, but easy to forget in practice
- **Standard hash-based indices only work for equality queries**
  - Although more powerful alternatives exist sometimes
  - E.g., Oracles range scans for queries using < or >
- **Index is only useful if there are many non-matching rows**
  - E.g., index on the EMPLOYEE.Sex column is probably not a good idea
- **INSERT INTO, UPDATE, and DELETE FROM becomes slower with an index**
  - Inserting now also needs to keep the index up-to-date
- **Index takes up disk space**
  - Non-trivial amount for large tables

Key Takeaways

Transactions are bundles of statements that should be executed in an all-or-nothing fashion

- **ACID** (Atomicity, Consistency, Isolation, Durability)
- **Database Isolation Levels**

Views (especially **materialized views**) are a way to improve the join performance of a database without compromising on normalization

Indices are another common way to improve the query performance of a database

- Especially remember the concept of a **full table scan**