CS 348

Shale Craig

February 26, 2014

Contents

Part I

Midterm Content

Chapter 1

Introduction

1.1 Motivation

We're motivated to develop corporate databases in an increasingly information-oriented society. How do we do this? What does the fox say? Nobody knows.

1.2 Data Management

Objective: to represent (a part of) the world usefully while *abstracting* away the details.

Intension of data conceptually describes the schema of a database (or data).
`EMPLOYEE(SIN, NAME, ADDRESS, BIRTHDATE, SALARY`

Extension of data is actually instances of data.

```
(1234, "Foo Bar", "123 Bar Street", Jan 1, 197
```

1.3 Components of a Database System

- Hardware
- Software
  - Application Programs
  - Utility programs
  - OS
- Database Management System (DBMS)

  Dictates data structure with simple CRUD support through variety of concurrent access methods.

1. *Concurrency Control* exists so queries are externally consistent.
2. *Security* exists to protect data from unauthorized access (password controlled, etc).
3. *Integrity* exists as constraints to ensure that data in the database is accurate and meaningful.
4. *Recovery from Failures* protects the database from dying.

1.4 Data

- Is the content of the knowledge of the organization.
- *Logical Files* as seen by application programmers - deals with the layout.
- *Physical Files* as seen by system programmers - performance.
- Very often, the two types of files are related.

1.5 DBMS Architecture

Exists within a dynamic environment where programs are highly linked, which leads to high maintenance cost.

We'd like to introduce independence between data and the application programs that use them, so either can be changed.

We'd like to maintain the ability to make logical changes (to the data) explicitly, without physical changes (storage media) affecting it.

Data Definition Language specifies the schemas and their mappings. (DDL)

Data Dictionary is the result of compilation of DDL statements in a schema.

Data Manipulation Language is the commands that are issued to the host program. (DML)

Query Language is a language for interactive data manipulation.

Chapter 2

Data Models

Data Models are guidelines and structure for organizing value-based or object-based data and executing operations within constraints.

2.1 Entity Relationship Model (ERM)

Entity Sets are a thing or object that can be distinctly identified (an *entity*) grouped into sets.

This section glosses over terms such as “entity”, “relationship”, and “relationship type”.

This is the notation that goes as follows:

Librarian (Emp#, Name, Salary, Addr, Allowance, Union, LibraryName)

This indicates that librarians have the primary key *Emp#*, and the attributes specified afterwards (Name, Salary, etc).

2.2 Entity-Relationship Diagram (RD)

This is an extension of UML that we can quickly convert to something actually useful.

2.2.1 Structure

This is a UML-like graph. Since you probably don't know about the dialect of UML that this professor wants us to know, here's a quick primer of what you need to know:

Entity Types are specified by rectangles. These connect to Relationship Types and Attributes.

Relationship Types are specified by diamonds. These connect $\geq$ two Entity Types.

Attributes are specified by ellipses. These connect to single Entity Types.

2.2.2 Constraints

We can establish constraints on these as follows:

Primary Keys

There are two types of keys:

Candidate Keys are minimal sets of attributes whose values identify an entity at all times.

Primary Keys are the main way of identifying an entity. They must be a candidate key.

Cardinality

We can force the number of entities (i.e. the cardinality of the ones involved) using UML-like (x, y) notation on relationships.

Generally, it's expressed as (min, max). The restrictions are $0 \leq x$, and y can be any number or a $*$[1].

The restrictions are written next to the corresponding entity.

Existence Constraints

By designing our ERD properly, we can create an entity A which is totally dependent if every existence of A is always associated with another entity through a relationship.

[1] The star indicates "any".

2.2.3 Extended ERM (EERM)

Generalization and Specialization

We can extend what ERM specifies by using inheritance.

We can express the similar properties using the concept of **generalization**[2].

By adding a tree-structure, we can specify our ERDs with a tree-like structure. Parent entities are connected to triangle blocks that say "IS-A" to their children.

Aggregation

Supposing that we want to construct relationships to relationships[3]. In this case, we can make a box around the elements from the first relation (almost like they're an "entity"), and connect the relationship in the box to the external relationships desired.

[2] He tends to enjoy the "is-a" concept more than the phrase "generalization". I like the word "inheritance" more.

[3] An example of this is that a *Student* may *Participate* in a *Project*. There is a relationship *Eval* between the *Student*, *Participate*, and *Project* and a *Report*. This is an example of this idea.

Chapter 3

Storage Systems and File Structures

3.1 Introduction

Data is stored on primary or secondary storage.

3.1.1 Primary Storage

Primary storage tends to be faster (caches, dynamic random access memory (DRAM)); it's fast but expensive.

3.1.2 Secondary Storage

Secondary storage tends to be larger than large, but it's also very slow.

3.1.3 Disk Storage Devices

Disks are the most common type of secondary storage, and can hold terabytes.

They are constructed of *disk packs* of *magnetic disks* connected to a rotating spindle. The disks have concentric circular *tracks* on each surface. Tracks with the same diameter form a *cylinder*. Each track is divided into equal size units called *blocks* or *pages*.

Pages are moved into main memory on demand. Since the access time is 30ms, and the CPU takes nano-seconds to process things, I/O is the bottleneck.

3.2 Reducing Latency and Page Accesse

We store pages containing related information near to each other since applications are likely to read them next[1].

3.2.1 Accessing Data Through a Cache

Files are a sequence of records stored on disk blocks.

Records can be **fixed-length** or **variable** length, and can span only one block.

Physical disk blocks allocated to hold records of a file can be *contiguous*, *linked*, or *indexed*.

[1] Is this called the principle of locality?

3.2.2 Ordered Files

Ordered files (or *sequential files*) are records kept sorted by the values of an ordering field.

Insertion can be expensive, so some implementations use an *overflow file* for new records to improve insertion. This is periodically merged with the ordered file.

Binary searches are used within a file to find a record with a given value of the ordering field.

Chapter 4

Indexing Structures for Files

4.1 Introduction

Given an attribute value, we want to retrieve all records that match that attribute. Indexes are great, since when looking up according to an index we can get optimized lookup[1]. This speedup is great, but it comes at cost of an index file.

4.2 Types of Indexes

There are a few types of indexes:

[1] We don't have optimized lookup for a non-index.

Search Key is the set of attributes on which an index is built and not the one that's always built.

Primary Index is the index that the index for a set of entities is built around. Usually, this is the search key[2].

Ordered is a term referring to the search key ordering in the index file. Often they are ordered, but if they're not the primary index, they are *unordered*.

4.3 B$^+$ Trees

B$^+$ trees are dynamic index-based data structures that are made up index and data blocks. They are a special type of tree optimized to reduce the number of page misses.

4.3.1 Specification

For a B$^+$ tree of order m and a maximum data node size of d[3], we work under the constraints:

- All values are in leaf ("data") nodes.
- All leaves are on the same level.

[2] In the cases where the primary index isn't the search key, the search key is known as the secondary index.

[3] Often, $m \neq d$.

- With the exception of the root, every node has $[\lfloor \frac{m-1}{2} \rfloor, m - 1]$ keys, which are sorted in ascending order.
- An internal ("index") node with k keys has $k + 1$ pointers to children on the next level[4].
- Data nodes have $[\lfloor \frac{d}{2} \rfloor, d]$ sorted records, and a pointer to the next and previous data node.

Modifying this tree runs in logarithmic time.

The data nodes of a B^+ tree contain pointers to the next and previous data node for efficient iteration. The code to update this the next and previous data node would be found in the functions *split* and *merge*.

4.3.2 Insertion

Insertion can be expressed as the following pseudocode:

```
insert(node n, btree b):
    node = b.findLeaf(n)
    while (true):
        node.insert(n)
        if (!node.isOverflow())
            return
        if (node.isRoot()):
            split(node)
            return
```

[4]The children correspond to the partition induced on the key space by the k keys.

```
n = split(node)
node = n.getParent;
```

Basically, we insert into the node. If we don't have enough room inside the data node, we split it (and insert the split nodes into the parent structure).

Split - Data Node

In the case that we call split on a data (leaf) node n_j, we create a new node n_{j+1} that will contain half the records of the old root node. For an odd number of records, keep the extra record in node n_j. Promote[5] the largest key value to the parent index node.

Split - Index Node

In the case that we call split on a index node with keys $k_j \rightarrow k_{j+n}$, we partition the values into $[k_j \cdots k_{j+\lfloor \frac{n}{2} \rfloor - 1}]$ and $[k_{j+\lfloor \frac{n}{2} \rfloor + 1} \cdots k_{j+n}]$. For an odd number of records, keep the extra record in node n_j. Move[6] the key k_j to the parent node, and keep pointers to the left and right nodes in the data structure.

[5] Just clone the value, don't remove it from the data node.

[6] Move, not clone.

4.3.3 Deletion

Deletion seems like a pretty annoying algorithm because leaf height needs to be maintained, but it's surprisingly simple:

```
def delete(index idx, bTree b):
    node toDelete = b.find(idx)
    entry = toDelete.getEntry()
    while (entry != null):
        node parent = entry.parent
        parent.remove(entry)
        if (!parent.isUnderFlow()):
            return
        if (parent.isRoot()):
            collapseRoot()
        if (!parent.leftNeighbor.isMinimal()):
            redistribute(parent, parent.leftNe
        merge(parent, parent.leftNeighbor);
        entry = parent.getEntry()
```

Redistribute - Data

Redistribute records as evenly as possible between two siblings, updating the parent key accordingly.

Redistribute - Index

Redistribute indexes as evenly as possible between two siblings, updating the parent key accordingly.

Merge - Data

Merge the two data blocks into one (move the records from right to left), and delete the parent entry that divided the two.

Merge - Index

One sibling has below minimum, while the other has minimum. Merge the two, and delete the parent key separating the two.

Chapter 5

Relational Model

Relational models are used to design and model relational databases, foo'!

5.1 Terminology

We only have one data structuring tool - a **relation**. We can express relations as $D_1 \times \cdots \times D_n = \{\langle a_1, \cdots, a_n \rangle | \forall i : a_i \in D_i\}$. We call $r = \langle a_1, \cdots a_n \rangle$ a relation on n-sets; i.e. r is a set of **tuples** (often called **rows**). We call D_j the j^{th} domain of r, where r is of degree n.

When writing databases, we use the **Closed World Assumption** to govern what exists - the assumption that everything not currently know to be true is false[1].

[1] i.e. if I don't know about it, it must not exist.

Relational schemes define the composition (intension) of a relation.

Domains are the limited scope that an attribute is valid under.

5.2 Key Propagation

For many:many relationships (i.e. many `doctor`s may be the `attendingDoctor` for a many `patient`s), it's a good idea to keep keys joining the two in a separate relation.

5.3 Constraints

5.3.1 Inherent Constraints

Inside a relationship, there are a few basic constraints:

- We need to have a *Candidate Key* that is a minimal set of attributes to uniquely identify tuples. This candidate key cannot be repeated.
- We need to have a *Primary Key* that cannot be updated, duplicated, nor contain nulls (**entity integrity rule**)

Domain Constraints

In core SQL-99, we can limit attribute values using the `check` constraint or creating a new domain. The domain

is used in schema declarations, and is a schema element:

```
CREATE DOMAIN Grades CHAR(1)
    CHECK (VALUE IN ('A', 'B', 'C', 'D', 'F'))
```

We can use this "type" later:

```
create table Transcript (
    ...,
    grade Grades,
    ...
)
```

Foreign Key (Referential Integrity) Constraints

Given that a set of attributes $FK \subseteq R_1$ is a *foreign key* that references R_2, we have two constraints to satisfy:

1. The attributes of FK are defined on the same domains as the primary key PK of R_2.
2. A value of FK either occurs as a value of PK, or FK is null.

We say that R_1 references R_2[2]

[2]He took extra time to point out that R_1 is the *referencing* relation, and that R_2 is the *referenced* relation with respect to this foreign key constraint.

5.4 Tables

We use tables as our basic type of represent relations.
The SQL syntax is as follows:

```
create table <tname> (
    (columnDec)+
    [, (tableConstraint)]
);
columnDec =
    <colName> <colDataType> [default <value>][(
colConstraint = {
    not null |
    [constraint <name>] unique |
    primary key |
    check (search_cond) |
    references <tname> [<colName>] [(refEffect)
}
tableConstraint = [constraint name] {
    UNIQUE (<colName>+) |
    foreign key (<colName>+) references <tname>
}
refEffect = on {update | delete} (effect)
effect = {
    set null |
    no action (restrict) |
    cascade |
    set default
}
```