Databases Notes
http://www.matthew.ath.cx/notes/

Matthew Johnson
Trinity Hall
mjj29-notes@srcf.ucam.org

May 27, 2003

1 Intro

This course has been rewritten in the last year or so, and the notes list questions that are still appropriate to the course. Comments on lectures and course welcome. Extra notes & details on the web-site:

http://www.cl.cam.ac.uk/Teaching/2002/Databases.

Books: See notes, also: Stonebraker, Brown *Object-relational DBMSs* - Stonebraker wrote PostgreSQL.

2 What this is about

Databases is not just about storing data, its about organising it, so that you can easily retrieve the data you want. *Reliability* is also important in a corporate environment, as is *consistency* of data. The last is to do with when you have constraints on the data, keeping those constraints. It would be nice to have the database manage this for you. All of the above are inter-related.

The Stanford course contains the following elements

- Modelling & Design of databases
- Programming - SQL queries on the database
- DBMS implementation

we look at the first two items, and we will be playing around in the lectures with SQL queries.

3 What is a database system?

A database is just a large, integrated, collection of data. Databases model real world enterprises. The Database Management System (DBMS) is the software system designed to access and manage the data. We are designing systems to hold very large amounts of data - for example databases in particle accelerators generate between 10Mb and 1Gb of data per second.

These constrains make it very difficult to get efficient access to this amount of data. Mainly the amounts of data that are being stored won't fit in main memory - however, with advances in memory there are a lot of applications that can be stored.

The features we want to implement are:

- efficient access to the data
- concurrent access to the data
- secure and atomic access to the data

- All applied over a possibly very large dataset.

3.1 Why should we study Databases?

Everybody needs databases - recently the need has exploded, and there is a lot of money involved. There are also lots of interesting and unsolved research issues, despite the age of the field. For example, storage of web data, and genomic data that doesn't match the relational database model. One of the other main areas is in data mining. This is the technique of taking large amounts of less connected data, and trying to infer other information through statistical techniques.

There are also lots of connections with other areas of Computer Science, you will have to know about:

- Programming languages and software engineering
- Data structures and algorithms
- Logic, discrete maths, computation theory
- Systems problems: concurrency, operating systems, file organisation, networks, distributed systems.....

Note: Currently, at least 80% of the world's data is not in a database - because it doesn't fit the model of databases; for example scientific and personal data - and of course the Web.

3.2 Coding Databases?

For a lot of things, storing data in a flat file and using grep, sed etc - or with some perl is fine, so why shouldn't we do this all the time?

When things get more complicated, however, there are lots of problems:

3.2.1 Data Organisation

We may add fields in our text file that duplicate other data stores, or are duplicated within our file - we really want to just be able to reference another data store.

3.2.2 Efficiently

There are functions that we do regularly, which we want to happen quickly. These queries can be quite complicated, and in large data sets can run slowly. A generic system for queries is useful, as its easily extendible.

3.2.3 Concurrency & Reliability

If you want to allow multiple access to your data, how do you manage keeping consistency if two people edit at the same time? If a system crashes, can you guarantee a consistent state?

A DBMS solves this problem.

4 Database System Architecture

It is common to describe databases in two ways. The *logical level* is what users see, the query language, or the programming API presented to the users. The *physical level* is to do with actual implementation of the storage and indexing systems. It is traditional to split the logical level into two: the overall design, and the different views that various users get to see. A *Schema* is a description of the database, and these can be at both the conceptual level, or the physical level.

The three level architecture to databases is:

- an *internal schema*, that is a description of the physical level.
- a *conceptual schema*, which is a complete abstraction away from the physical level, and doesn't describe any of the implementation. This would use a high level data description language.
- possibly many *external schema*, representing different users' views.

There should be independence between the physical and logical layers - it should be possible to change the schema at one level, without affecting higher levels. Logical data independence is the ability to change the conceptual schema without changing the user schemas, and physical data independence is the capacity to change the physical implementation without affecting either of the higher layers.

In actual databases it is very important to be able to change, add and remove tables, without affecting the use of the system by any of the users.

5 Relational Model of Data

We study the relational model of data for several reasons. Firstly, its the model that all the major database vendors use - Microsoft, Oracle, IBM etc. SQL is the industrial standard of the relational model - however, it has been rewritten several times, and most of the commercial systems have extended the standard themselves. The relational model itself is quite simple - which is why people have extended SQL to perform more complex things.

After databases were first written in the 60s, it was realised a better system for storing the data was needed. Around 1970 E.F. Codd proposed the relational model, and it was prototyped for the next 10 years before being implemented. Since then this has been the main database model used. Lots of optimisations and transactional support were also developed during this time (IBM System R - relational model is very high level, so hard to implement efficiently - System R managed it, and lots of their optimisations are still the main tricks used today).

5.1 Basics

A *relational database* is a collection of relations. Each relations consists of an *instance* - an actual table, with rows and columns - and also a *schema*, which specifies the name of the relation, and also the name and type of each column. The schema is a meta-description of the language. Normally written:

```
Students(sid:string, name:string, login:string, age:integer)
```

In this case `Students` is the relation name, `sid` is a field name and `string` is a *domain*. The domain is the type of the field, and we assume they are atomic - although this restriction is sometimes dropped in practice. The *arity* of a relation is the number of fields, or columns, and the *cardinality* of the relation is the number of rows (items in the relation).

More formally, a relation can be defines as a set of tuples (rows). There is no ordering over the tuples, and should have *no duplicate rows*. This is not the default in current SQL system. Formally the relation $R = R(a_1 : \tau_1, ..., A_n : \tau_n)$ can be defined as $R \subseteq \tau_1 \times ... \times \tau_n$.

5.2 Keys

Given a relation $R = R(a_1 : \tau_1, ..., A_n : \tau_n)$ a *key* is a subset of fields $K \subseteq \{A_1, ..., A_n\}$. We annotate the schema by putting a bar over the keys in the schema $R = R(\overline{a_1 : \tau_1}, ..., A_n : \tau_n)$

5.3 SQL

This is the ubiquitous language for relational databases. It was originally standardised in 1992, and SQL/92 should run on any of the major databases with the same results. Part of SQL is a table definition language, which we can use to define and modify tables, as well as a query language for retrieving data.

5.3.1 Simple SQL Commands

```
CREATE TABLE students(sid char(20), name char(20),
      login char(10), age integer);
DROP TABLE students;
ALTER TABLE students ADD COLUMN matric integer;
ALTER TABLE students DROP COLUMN matric;
INSERT INTO students (sid, name) VALUES ("1006","Britney");
DELETE FROM students WHERE sid = "1006";
SELECT * FROM students;
SELECT 4*4;
```

6 Building relational schema

The E/R model of data can easily be mapped to the relational model, although you will have to flatten the relations to schemas, and weak items (ones without a unique key) must be combined with strong items to generate the schema.

Usually you will end up with multiple tables to store all your data. This will potentially be inefficient in doing queries that span all the tables - naïvely you will have to walk over all of the (potentially large) tables to answer a query.

7 Relational Algebra and Query Languages

Query languages allow the manipulation and retrieval of data from a database. The relational model supports simple, powerful query languages, mostly with a strong formal foundation. They are not, however, necessarily programming languages - it was felt that making query languages Turing complete was a bad idea, since you don't want queries that can loop forever. Custom extensions of SQL, however, and the new SQL standard are Turing complete.

7.1 Formal relational languages

There are two formal query languages that were introduced by Codd, *Relational Algebra*, which is an operational model, that expresses execution plans, and *Relational Calculus* which is logical, declarative, language. Most SQL languages are implemented using relational algebra underneath. These two languages have equivalent expressive power. Anything that is equivalent to those is called *Relationally Complete*.

7.2 A Query

A query is applied to relational instances, and the result is also a relational instance. The Schema of relations are fixed, and the query is checked against that schema (c.f. type checking). If its a valid query, it is then executed against a valid instance. The result schema (type) can be determined from the query.

There are 5 basic operator classes:

7.2.1 Selection - σ

Selects a subset of rows. We have a condition c, involving the attributes of a relation R_1. The result $R_2 = \sigma_c(R_1)$. The schema of the result is exactly the same of the input instance, and there are no duplicates (it is a relation). We can also use the result as the input of another operator - to compose several operators.

The property that everything returns a relation is why SQL returns things like 4×4 as a relation.

7.2.2 Projection - π

This deletes the fields (columns) that are not in the projection list. $R_1 = \pi_A(R_2)$. You really should remove duplicates within the reduced set of columns. However, most real systems don't do this unless you ask.

7.2.3 Renaming - ρ

This returns a relation instance identical to the input, but with one of the fields (columns) renamed.

7.2.4 Set operations

Since relations are sets, we will allow use of the familiar set theory operators ($\cup$, $\cap$, $-$). However, you cannot take the union of two arbitrary relations. They must operate on relations of all the same type (column number, name & domain).

7.2.5 Cartesian Products $\times$

$A \times B$ - concatenate every row of A with every row of B. This comes into difficulty when A and B share column names. There are several choices to deal with it, but we can assume it just generates suffixes for duplicate column names.

7.3 Composite Operators

7.3.1 Theta Join - $\bowtie$

This is theoretically a derived operator, but it is very often useful, so we define it separately. $R_1 \bowtie_C R_2 \equiv \sigma_C(R_1 \times R_2)$. It is the rows of a Cartesian product that satisfy some condition.

The resultant schema is the same as for the Cartesian products, and is sometimes known as a *conditional join*.

7.3.2 Equi-join

This is the special case of a Theta join where the condition is equality of field names (e.g., two tables with a sid, select all the data from both, where the sid is the same in both). A *Natural Join* is an equi-join on all common fields. This is just written $A \bowtie B$.

The natural join operator appears very often in real life, and is always implemented directly by the query language. This is because the initial Cartesian product produces a very large table, and the filter you apply will almost always remove most of the result of the product, so you implement it directly. In fact it is such a common operator that it is the topic of most of the research into improving efficiency.

7.3.3 Division $/$

This is useful for queries like "Find sailors who have reserves all the boats". if A has fields x, y and B has field y, then A/B is the relation of xs such that for every y in B there is a line x, y in A - $xs = \{x | \forall y \in B, \exists (x, y) \in A\}$.

We can code this using the basic operators by: doing $\pi_x(A) - \pi_x((\pi_x(A) \times B) - A)$. This is quite a complex program to implement something we probably want to do.

7.4 Limitations

There are limitations in this algebra. You cannot do queries which can iterate, or recurse, an arbitrary number of times.

8 Relational Calculus

We write relational calculus queries using set notation:

$\{\langle I, N, R, A\rangle | \langle I, N, R, A\rangle \in Sailors \wedge R\rangle 7\}$. The first condition (membership of the tuple) specifies which table to get the data from, the second condition applies a restriction.

projection: $\{\langle I, N\rangle | \exists R, A. \langle I, N, R, A\rangle \in Sailors\}$

In general it has the form $\{\langle x_1, ..., x_n\rangle | F(x_1, ..., x_n)\}$

Formulas are defined as:

Atomic formulae: $a ::=$

$\langle x_1, ..., x_n\rangle \in R$
$x_i \ binop \ x_j$
$c \ binop \ x_j$
$x_i \ binop \ c$
$unop \ x_j$
$unop \ c$

Domain Relational Calculus formulae: $P, Q ::=$

a
$\neg P$
$P \vee Q$
$P \wedge Q$
$\exists x.P$
$\forall x.P$

$\exists$ and $\forall$ are **binders** for x.

Joins can be simulated by $\exists$, such as in this :

$$\{\langle N\rangle | \langle I, N, R, A\rangle \in Sailors \wedge R\rangle 7 \wedge \exists SI, BI, D.$$
$$(\langle SI, BI, D\rangle \in Reserves \wedge I = SI \wedge BI = 103)\}$$

Which means "Names of sailors above rating 7, who have reserved boat 103".

8.1 Equivalence with relational algebra

This was first considered by Codd in 1972. He introduces the notion of *relational completeness* - if a language can code any query in relational algebra then it is relationally complete.

To prove that relational calculus is equivalent to relational algebra, we must show how to encode relational algebra in it.

If we assume that some table/schema e is represented in relational calculus as $[[e]] = \{\langle x_1, ..., x_n\rangle | F\}$, then we can do a select over it by adding a condition: $[[\sigma_C(e)]] = \{\langle x_1, ..., x_n\rangle | F^{C'}\}$. C' is obtained by translating all the fields from the schema in the select condition C to variables in the relational calculus. You can similarly perform projection.

The mapping the other way is more tricky. There are queries that can have infinite answers depending on the relational encoding. What's more, the problem of knowing whether an encoding will be finite is undecidable. We can, however, define a restricted subset which is guaranteed to be *safe* (always finite).

9 SQL & SQL/92

SQL originated from the IBM System R project. It was then standardised in 86, 89, 92, and 1999 - which is still being written. The original standardisation was by ANSI, but the current standard (SQL/92) is an ANSI/ISO standard. Off the shelf database products implement the SQL/92 standard. However, the standard specifies 3 different levels of compliance, and products also add their own extensions. This means most of the actual databases in use are not SQL/92 compliant, and hence not portable.

SQL is a comprehensive database language, so it contains a Data Definition Language (DDL) - for defining schema, and a Data Manipulation Language (DML) - for inserting, deleting, modifying and retrieving data etc. It also contains other features we aren't going to look at such as security and dynamic features.

9.1 SQL notes

The use of `*` in the query `SELECT * FROM sailors;` is considered bad form, since the schema for `sailors` can change over time, and you should specify all the fields you want.

SQL also defines relations as multi-sets, not sets (i.e. you can have duplicate rows). This is mainly for efficiency reasons.

When specifying multiple tables, you bind the tuples to variables. E.g. `SELECT S.sname FROM Sailors S, Reserves R WHERE S.sid=R.sid AND R.bid=103;` Using `Sailors.sid` is considered bad form.

SELECT Syntax: `SELECT [DISTINCT] <target-list> FROM <relation-list> WHERE <qualification>`

`DISTINCT`	Removes duplicates
`<target-list>`	List of attributes we want in the result
`<relation-list>`	Relations we want to take the Cartesian product of
`<qualification>`	Boolean expression to filter rows on

The SQL `SELECT DISTINCT a1, ..., an FROM R1, ..., RN WHERE b` is equivalent to the relational algebra: $\pi_{a_1,...,a_n}(\sigma_b(R_1 \times ... \times R_n))$ (assuming no duplicates).

When comparing rows in the same table, you can reference the same table twice. E.g.: `SELECT s1.name, s2.name FROM sailors s1, sailors s2 WHERE s1.rating=2*s2.rating-4`

9.2 Operations

SQL provides features for matching strings using basic pattern matching. It also provides all the normal arithmetic operators for use in queries, but with the caveat that it always returns these things as relations.

SQL provides the key words `UNION`, `INTERSECT` & `IN`, which are the standard set theory operations. You can use these to nest queries within each other.

```
SELECT S.sname FROM sailors S WHERE S.sid IN
(SELECT R.sid FROM reserves R WHERE R.bid=103)
```

However, programs like MySQL don't support some of these features (like nesting queries).

10 Integrity Constraints

An *Integrity Constraint* (IC) is a condition that must be true for any instance of the database, e.g. domain constraints on a field. ICs are specified in the schema, and should be checked whenever the relation is modified. A *legal instance* of a relation is one that satisfies all the constraints. We would like the DBMS to check it for us, and enforce validity on all relations and modifications.

10.1 key constraints

A set of fields is a (candidate) key for a relation if no two distinct tuples can have the same value in all key fields, nor any subset of the key. A primary key is one picked by the DBA. SQL has the key words `PRIMARY KEY` and `UNIQUE` for the candidate keys.

A relation may contain a field corresponding to the primary key from another relation, and this acts as a pointer to a tuple in the other relation. This is called a *foreign key*. SQL allows us to spell this out using the `FOREIGN KEY` keyword. E.g:

```
CREATE TABLE Reserves (sid INTEGER, bid INTEGER,
                       date DATE, PRIMARY KEY (sid,bid,date),
                       FOREIGN KEY(sid) REFERENCES Sailors)
```

This then allows the DBMS to enforce the foreign key pointing to valid tuples in the other table.

10.2 Semantic Constraints

Another type of constraints called semantic constraints, which you can use to restrict what values are valid in a field.

11 Database Schema design

Not all designs are equally good. For example:

```
Data(sid,sname,address,cid,cname,grade)
```

compared to:

```
Student(sid,sname,address)
Course(cid,cname)
Enrolled(sid,cid,grade)
```

Why is the second set of schema better than the first one? Using the first one you can get redundant information - for example the student name, id and address will be replicated for every course they take, and similarly the course id/name. Secondly, when trying to update one of these duplicated entries you need to update all of these duplicates, and have to be careful not to miss any. You also run into the problem of adding extra courses, when you don't yet have any students taking the course.

Redundancy is the cause of most of the above problems, along with the various anomalies noted above. We could attempt to solve it using NULL values, and that works in some cases - but not all. In the example we wanted to use `sid` & `cid` as a key - setting one of those to NULL is a bad idea.

We can fix all of the problems by using the second schema above, containing 3 tables. A relation for courses, a relation for students, and a relation linking the two first relations, and storing the grade for that student in that course. You do have to consider a few extra problems from decomposing your relations into several smaller ones. We have just generated the second schema out of the air, but there are systematic ways of decomposing your relations to smaller ones, specifying when you should stop, and ensuring you don't lose any semantic information.

11.1 Functional Dependencies

We have specified that if two fields have the same key, they should have the same other fields. In the case of the example above, we can say that the `sid` determines the `address`. This is called a *functional dependency*, and is written `sid` → `address`. In general a functional dependency $X \to Y$ is simply a pair of sets of field names.

In the example: $\begin{array}{rcl} \texttt{sid} & \rightarrow & \texttt{sname,address} \\ \texttt{cid} & \rightarrow & \texttt{cname} \\ \texttt{sid,cid} & \rightarrow & \texttt{grade} \end{array}$

These are semantic assertions we make about the data. These are requirements we get from the client, rather than ones we can get from analysing the data (although we can check that the data satisfied the FDs).

11.1.1 Properties of Functional Dependencies

Functional Dependencies are transitive, so if $X \rightarrow Y$ and $Y \rightarrow Z$ then $X \rightarrow Z$. The closure of F is the set of all FDs implied by F, ie $F^+ \equiv \{X \rightarrow Y | F \vdash X \rightarrow Y\}$. F^+ can be very big.

If in a relation $R(A_1, ..., A_n)$, $X \subset \{A_1, ..., A_n\}$ and $X \rightarrow \{A_1, ..., A_n\}$, then X is a *candidate key* for the relation (as long as there is not a smaller subset for which that is true).

FDs are also reflexive (`address` $\rightarrow$ `address`), transitive (as we said before) and also are augmentative: if $F \vdash X \rightarrow Y$ then $F \vdash X, W \rightarrow Y, W$. As a consequence of these, we can take the union - $F \vdash X \rightarrow Y$ and $F \vdash X \rightarrow Z$ then $F \vdash X \rightarrow Y, Z$, and similarly the reverse, we can always imply just a subset.

11.1.2 Equivalence

If we have two sets of FDs, they are equivalent if $F^+ = G^+$, however, since F^+ can be huge, we prefer to look for smaller sets. We define an FD to be *minimal* if:

1. Every FD in F is $X \rightarrow A$, where A is a single attribute
2. $\forall X \rightarrow A \in F, F - \{X \rightarrow A\} \neq F$
3. The left hand side is as simple as possible.

The axioms we had above 'do the right thing' because they imply soundness (if we have deduced a rule using the axioms from F then it is true whenever F is) and completeness (if a rule is true in a relation where F holds, you can deduce $F \vdash X \rightarrow Y$). Soundness is a lot easier to prove than completeness!

Calculating the closure can be exponential in complexity. If we are just checking that an attribute is in the closure, we can calculate X^+, the attribute closure, which is only quadratic in complexity. In fact a linear algorithm is known.

12 Decomposing Relations

We can't just decompose any relations, and when we do, there may be extra problems introduced.

Formally, a decomposition of a relation R is a collection of subsets of R, such that $R = R_1 \cup \ldots \cup R_k$. Relations $R_1 \ldots R_k$ need not be disjoint. IE, we can't throw away any attribute, or arbitrarily introduce them, but there can be overlaps.

Not all decompositions are 'good'. In the previous example we could have decomposed into `R1(sid,sname,address)` and `R2(cid,cname,grade)`, but here we have lost the relationship between students and courses. We therefore define a *lossless-join decomposition* of R if for every instance r of R: $\pi_{R_1}(r) \bowtie \ldots \bowtie \pi_{R_K}(r) = r$. If this is not satisfied, you generate extra rows of data, because of the Cartesian product, which aren't in the original table.

This is still not enough to produce a 'good' decomposition. We also need to consider functional dependencies over the relations. We don't want checking for functional dependencies on update to require joins of tables, so we need to be able to enforce the dependencies on the individual decomposed relations, and have that equivalent to the FDs on the original relation. We can waive this condition if necessary - it is merely an efficiency concern, not semantic integrity.

This is formally written, as you might expect, as $F^+ = (F_{R_1} \cup \ldots \cup F_{R_K})^+$. This seems like it could be a very expensive algorithm to compute, however, there is a polynomial time algorithm to compute this in the case of dependency preservation.

12.1 Boyce-Codd Normal Form

A relation R with FDs F is said to be in *Boyce-Codd Normal Form* (*BCNF*) if for all $A \to X \in F^+$ then either $A \in X$ or X is a super-key for R. Intuitively we can say that a decomposition of a relation is in BCNF if all of the decomposed relations are in BCNF.

Boyce-Codd normal form doesn't guarantee functional dependency, some times we will have to drop either the FD condition or the BCNF.

12.2 Third Normal Form

3NF is a weaker condition than BCNF, and adds the condition that A could be a member of a candidate key for R.

This was derived from First and Second normal form, which were to do with atomicity of domains, and partial functional dependencies. These normal forms are progressively stricter subsets of previous ones.

Even after defining all these conditions on a decomposition, we can still generate tables with redundant data and the accompanying update anomalies. We need to extend the theory to capture this for redundancy.

13 Further Relational Algebra and SQL

There are several places that the relational algebra and SQL don't match up. Firstly, SQL doesn't remove duplicate rows - we are dealing with *multi-sets* (*bags*), not sets. We could use the `DISTINCT` option in SQL, but it's better to extend relational algebra to bags. Bags are un-ordered groups, like sets, but allow duplicates.

13.1 Bag Operators

The set theory operators can be applied to bags too:

- Bag Union, $\{1,2,1\} \cup \{1,2,3\} = \{1,1,1,2,2,3\}$
- Bag Intersection
-

Set theory operators have equivalence rules. Some of these still apply to bag theory operators, but some of them do not.

Now we have extended relational algebra to include bags, we need new operators to remove duplicates, and perform other things that don't apply to sets.

13.2 Duplicate elimination operator (δ)

$\delta(R)$ returns a relation with no duplicates (a *set*).

13.3 Ordering operator (τ)

SQL allows you to specify a sort order to the results returned - unlike sets and bags, which are un-ordered. We will add a sort operator to relational algebra.

$\tau_{L_1,\ldots,L_n}(R)$ returns a *list* of tuples of R, ordered by attributes $L_1,..,L_n$.

13.4 Extended projection

SQL allows us to use arbitrary arithmetic in queries. We can extend the projection operator π to allow the columns in the projection to be functions of other columns. (e.g. $\pi_{A+B,A,B}$).

Note that you can't in SQL just issue arithmetic commands at the top level, you can only use it in select queries (e.g. "`SELECT 2+2`" - which returns a relation just containing 4).

13.5 Aggregation

There are several SQL operators that will perform some calculation over a whole column, and just returns the result. For example, we can find the average of a whole column with `SELECT AVG(age) FROM sailors`. Other operators you can use are:

Operator	Purpose
SUM	Sums all the values in that column
AVG	Returns the average of all the values in that column
MIN	Returns the minimum value in that column
MAX	Returns the maximum value in that column
COUNT	Returns the number of rows in the relation

These operators are applied to all of the tuples in the relation. Sometimes we want to aggregate over several groups. For example "For each rating, find the average age". SQL has a `GROUP BY` operator to allow us to do this.

For relational algebra we will introduce γ for grouping and θ for aggregation. The aggregation can only be given as an argument to grouping, e.g.: $\gamma_{L_1,\theta(L_2)}(R)$ - where θ can be any of the aggregation operators.

13.6 NULL values

SQL allows us to specify *NULL* values, for situations when the value is either inapplicable (single, with a spouse column) or unknown (they haven't got a rating yet). This complicates things quite a lot, when using operators on values (e.g. is NULL>8? what is NULL OR true?). We need to carefully redefine our semantics, and use *three valued logic* to handle it.

Note that `NULL = NULL` is false, since "`=NULL`" always returns `NULL`. SQL has the special operator `IS`, which returns true.

13.7 Integrity Constraints Extensions

Since we have introduced NULL values, we have to add another Integrity Constraint, which is that no primary key value can be NULL.

13.8 Outer Join $^\circ\bowtie^\circ$

Normally when you perform a join, you remove tuples that don't match. Outer join keeps the tuples in, but inserts NULL for the unknown joins. There are, in fact, Left, Right and Full Outer Joins ($^\circ\bowtie, \bowtie^\circ, ^\circ\bowtie^\circ$), which pad NULLs on the right, left or both respectively.

13.9 Views

SQL allows you to define "virtual tables", which you can specify in other queries, and it will substitute in the definition of the view before calculating.

14 Object Database Models

So far our only response to redundancy is to decompose it into normalised forms. This can have the effect of splitting our notion of some entity across several tables. We often might want a *set* of attributes in a tuple. 1NF has only atomic attributes, however, so we can't be 1NF compliant with sets. The *nested relational model* (NF^2), where values in tuples can themselves be sets or tuples.

This means we have to introduce a type system, and also `NEST` and `UNNEST` operators, which splits the tuple into multiple rows, each containing one of the members of the set, or joins multiple rows together. The last description, of the NEST operator, is a little difficult. You can't do this generically - since fields you aren't nesting may have differing content. NEST is therefore only a partial operator.

14.1 Object Orientation

The main point is that we need support for complex data types to handle increasingly complex data. Its also the case that database access has moved from dedicated DBAs and SQL programmers, to people accessing database from programs. This has the effect that the things which you want to store are likely to be complex objects - which are not suitable for storage in a flat database.

There were two main splits about how to alter SQL. The OO database system, who wanted to throw away the relation structure, and concentrate on collections of objects, which led to ODMG, and the 3rd-gen database system, who wanted to add object features to the relational model. This led onto SQL:1999.

There is also XML to consider, which is different again.

14.2 Pure object databases

This was intended to be completely integrated with the programming language. The idea was to have a shared type system, persistent storage of objects - along with the query language from SQL.

14.2.1 Object Data Management Group (ODMG)

This was a collection of DBMS vendors, who collaborated to produce the Object Data Standard. They defined a language independent object model, definition language, query language and interchange format. They then provided bindings for various languages so that they could use the ODS from any of them, and also access the same data from any of them. They also kept compliant with CORBA.

the Object Definition Language (ODL) is a class-based schema, based on CORBA's IDL. It allows single inheritance, and multiple inheritance of interfaces (similar to Java). Extents are used to reference all the instances of the class, and also methods and attributes. Relationships are defined using *traversal paths*. You define in the class definition a relationship between this object and either a single other object, or a set of other objects. If both classes refer to the other with a set, then you have a many to many relationship.

One to one relationship:

```
class one {
   attribute.....
   .... ;
   relationship two r2
      inverse one::r1;
}

class two {
   attribute ...
   ... ;
   relationship one r1
      inverse two::r2
}
```

One to many relationship:

```
class one {
   attribute.....
   .... ;
   relationship set<two> r2
      inverse one::r1;
}

class two {
   attribute ...
   ... ;
   relationship one r1
      inverse two::r2
}
```

Many to many relationship:

```
class one {
   attribute.....
   .... ;
   relationship set<two> r2
      inverse one::r1;
}

class two {
   attribute ...
   ... ;
   relationship set<one> r1
      inverse two::r2
}
```

14.2.2 Java Language Bindings

Queries are done by instantiating a query object, and creating a query with "holes" - `query.create("select ... $1 ... $2")`, and then putting arbitrary Java objects into the holes `query.bind(new Object())`.

Most "ODMG-compliant" databases aren't - and the standard is very buggy. Sun have, in fact, deprecated using ODMG, and they are proposing their own Java-specific language.

14.3 SQL:1999

SQL:1999 has taken many of the features that commercial databases were implementing themselves, and has incorporated many of their object features, and also added some new ones. They have added a lot of the features SQL92 didn't have - arrays and rows as fields, proper recursive queries etc. They have allowed user defined data types, with methods, single inheritance, sub-typing and references to UDTs. Methods can be declared as fields of a table (you define the code later), which can side-effect the record it is called on. Type hierarchies with field inheritance and overloading.

14.4 Differences between OODBMS and ORDBMS

OODBMS aims to add database functionality to an OO programming language, ORDBMS starts from a relational DBMS, and address OO ideas. OODBMS aims for object-centric applications, where you have few long-lived objects and has complicated memory management issues. ORDBMS aims for large collections of data, with perhaps shorter transactions. This has complicated disc-access issues.

15 XML

Documents and databases have a lot of differences, yet the distinction is not well defined. There is a mid-ground between them, which is a *data format*. This is an example of *semi-structured data* - which is data which is schema-less, i.e. there is no separate description of the type or structure of the data. This was originally done as work on databases, but is now being applied to XML.

XML can be used to represent tuples and fields, by nesting the fields in tags in a tuple tag, and also for lists:

```
<person>
   <name> ... </name>
   <phone> ... </phone>
   ...
</person>

<addresses>
   <person> ... </person>
   <person> ... </person>
   ...
</addresses>
```

Databases can be easily flattened to text files, but you can organise them in several ways - some of which might aid different types of searches.

XML can be represented as a tree of nested attributes.

15.1 DTDs

You can write XML without a schema, but there are facilities to provide one, to aid interactions between software. One of these is the *Document Type Definition* (DTD). This allows you to specify what tags may appear within other tags, and what attributes tags can have.

Using DTDs as schema has several weaknesses. You can't provide very strict typing (only strings), unlike in databases, particularly in IDs and IDREFs. Also, the DTD doesn't specify what elements an ID can point to - unlike keys in databases.

People have therefore suggested *XML Schema*, however, that standard has become very large and unwieldy.

15.1.1 Domain-Specific DTDs

Lots of DTDs have been specified for various specific applications (WML, MathML, etc). This allows people to exchange documents in that format and easily check them. They are effectively grammars for the data you are sending.

15.2 Databases & XML

If we are storing data in this format, we are going to want to be able to query it. We could code it in Java, but we want a high level query language. W3C are developing a query language for XML, called XQuery.

There are also issues of concurrency - how to we manage finer grained concurrency than the whole file? We also want to bring in the ideas of keys and referential integrity from databases.

The other issue is one of data mining - if everything goes into XML, we want to be able to get information from it.

15.2.1 XQuery

One of the main parts of XQuery is XPATH. This is a language for specifying nodes of an XML tree. So, to get all of the `cname` nodes you would write `/mybank/cust/cname`. You could get just the contents of the fields with `/mybank/cust/cname/text()`, and you can also add predicates, and selection of attributes with `/mybank/acc[balance>50000]/@accno`.

You can wrap up XPATH statements in control blocks (*For Let Where Return (FLWR) expressions*), for example:

```
<foo> {
   for $b in /foo/bar
   where ($b/baz = "quux" ...)
   return "<xml string>$b</xml string>"
}
</foo>
```