Introduction to Database Theory

Pierre Senellart

ÉCOLE NORMALE SUPÉRIEURE | PSL RESEARCH UNIVERSITY PARIS

École de Printemps en Informatique Théorique, 2019

Data management

Numerous applications (standalone software, Web sites, etc.) need to manage data:

- Structure data useful to the application
- Store them in a persistent manner (data retained even when the application is not running)
- Efficiently query information within large data volumes
- Update data without violating some structural constraints
- Enable data access and updates by multiple users, possibly concurrently

Often, desirable to access the same data from several distinct applications, from distinct computers.

Role of a DBMS

Database Management System

Software that simplifies the design of applications that handle data, by providing a unified access to the functionalities required for data management, whatever the application.

Database

Collection of data (specific to a given application) managed by a DBMS

Features of DBMSs (1/2)

Physical independence. The user of a DBMS does not need to know how data are stored (in a file, on a raw partition, in a distributed filesystem, etc.); storage can be modified without impacting data access

Logical independence. It is possible to provide the user with a partial view of the data, corresponding to what he needs and is allowed to access

Ease of data access. Use of a declarative language describing queries and updates on the data, specifying the intent of a user rather than the way this will be implemented

Query optimization. Queries are automatically optimized to be implemented as efficiently as possible on the database

Features of DBMSs (2/2)

Logical integrity. The DBMS imposes constraints on data structure; every modification violating these constraints is denied

Physical integrity. The database remains in a coherent state, and data are durably preserved, even in case of software or hardware failure

Data sharing. Data are accessible by multiple users, concurrently, and these multiple and concurrent accesses cannot violate logical or physical data integrity

Standardization. The use of a DBMS is standardized, so that it may be possible to replace a DBMS with another without changing (in a major way) the code of the application

Major types of DBMSs

- **Relational (RDBMS).** Tables, complex queries (SQL), rich features
- **XML.** Trees, complex queries (XQuery), features similar to RDBMS
- **Graph/Triples.** Graph data, complex queries expressing graph navigation
- **Objects.** Complex data model, inspired by OOP
- **Documents.** Complex data, organized in documents, relatively simple queries and features
- **Key–Value.** Very basic data model, focus on performance
- **Column Stores.** Data model in between key–value and RDBMS; focus on iteration and aggregation on columns

Major types of DBMSs

Relational (RDBMS). Tables, complex queries (SQL), rich features

NoSQL:

XML. Trees, complex queries (XQuery), features similar to RDBMS

Graph/Triples. Graph data, complex queries expressing graph navigation

Objects. Complex data model, inspired by OOP

Documents. Complex data, organized in documents, relatively simple queries and features

Key–Value. Very basic data model, focus on performance

Column Stores. Data model in between key–value and RDBMS; focus on iteration and aggregation on columns

Classical relational DBMSs

- Based on the relational model: decomposition of data into relations (i.e., tables)
- A standard query language: SQL
- Data stored on disk
- Relations (tables) stored line after line
- Centralized system, with limited distribution possibilities

Database theory

- Well-established research area within computer science, concerned with foundational et theoretical aspects of data management
- Motivations:
  - provide through theoretical tools advances in the practice of data management
  - provide through the scope of data management advances in theoretical computer science
- The success of current-day DBMSs relies in particular on a theoretical result: Codd's theorem
- This lecture: High-level introduction to database theory, focusing on the relational model

Plan

Relational schema

We fix countably infinite sets:

- $\mathcal{L}$ of labels
- $\mathcal{V}$ of values
- $\mathcal{T}$ of types, s.t., $\forall \tau \in \mathcal{T}, \tau \subseteq \mathcal{V}$

Definition

A relation schema (of arity n) is an n-tuple $(A_1, \ldots, A_n)$ where each A_i (called an attribute) is a pair (L_i, τ_i) with $L_i \in \mathcal{L}$, $\tau_i \in \mathcal{T}$ and such that all L_i are distinct

Definition

A database schema is defined by a finite set of labels $L \subseteq \mathcal{L}$ (relation names), each label of L being mapped to a relation schema.

Example database schema

- Universe:
  - $\mathcal{L}$ the set of alphanumeric character strings starting with a letter
  - $\mathcal{V}$ the set of finite sequences of bits
  - $\mathcal{T}$ is formed of types such as `INTEGER` (representation as a sequence of bits of integers between -2^{31} and $2^{31}-1$), `REAL` (representation of floating-point numbers following IEEE 754), `TEXT` (UTF-8 representation of character strings), `DATE` (ISO8601 representation of dates), etc.
- Database schema formed of 2 relation names, `Guest` and `Reservation`
- `Guest`: $((\mathtt{id}, \mathtt{INTEGER}), (\mathtt{name}, \mathtt{TEXT}), (\mathtt{email}, \mathtt{TEXT}))$
- `Reservation`: $((\mathtt{id}, \mathtt{INTEGER}), (\mathtt{guest}, \mathtt{INTEGER}), (\mathtt{room}, \mathtt{INTEGER}), (\mathtt{arrival}, \mathtt{DATE}), (\mathtt{nights}, \mathtt{INTEGER}))$

Database

Definition

An instance of a relation schema $((L_1, \tau_1), \ldots, (L_n, \tau_n))$ (also called a relation on this schema) is a finite set $\{t_1, \ldots, t_k\}$ of tuples of the form $t_j = (v_{j1}, \ldots, v_{jn})$ with $\forall j \forall i \; v_{ji} \in \tau_i$.

Definition

An instance of a database schema (or, simply, a database on this schema) is a function that maps each relation name to an instance of the corresponding relation schema.

Note: Relation is used somewhat ambiguously to talk about a relation schema or an instance of a relation schema.

Example

Guest

id	name	email
1	John Smith	john.smith@gmail.com
2	Alice Black	alice@black.name
3	John Smith	john.smith@ens.fr

Reservation

id	guest	room	arrival	nights
1	1	504	2017-01-01	5
2	2	107	2017-01-10	3
3	3	302	2017-01-15	6
4	2	504	2017-01-15	2
5	2	107	2017-01-30	1

Some notation

- If $A = (L, \tau)$ is the ith attribute of a relation R, and t an n-tuple of an instance of R, we note $t[A]$ (or $t[L]$) the value of the ith component of t.
- Similarly, if $\mathcal{A}$ is a k-tuple of attributes among the n attributes of R, $t[\mathcal{A}]$ is the k-tuple formed from t by concatenating the $t[A]$ for A in $\mathcal{A}$.
- A tuple is an n-tuple for some n.

Variants: named and unnamed perspectives

The version presented considers the attributes of a relation are ordered and have a name. This is what best matches the way RDBMSs work, but not necessarily the most pleasant to reason on the relational model.

Named perspective. We forget the position of attributes, and consider they are uniquely identified by their names.

Unnamed perspective. We forget the name of attributes, and consider they are uniquely identified by their position. One uses notation such as $t[2]$ to access the value of the second attribute of a tuple.

No major impact, one will use one or the other depending on what is convenient.

Variant: bag semantics

- A relation instance is defined as a (finite) set of tuples. One can also consider a bag semantics of the relational model, where a relation instance is a multiset of tuples.
- This is what best matches how RDBMSs work...
- ... but most of relational database theory has been established for the set semantics, more convenient to work with
- We will mostly discuss the set semantics in this lecture

Variant: untyped version

- In implementations, attributes are always typed
- In models and theoretical results, one often abstracts attribute types away and considers each attribute has a universal type $\mathcal{V}$
- We will most often omit attribute types

Plan

The relational algebra

- Algebraic language to express queries
- A relational algebra expression produces a new relation from the database relations
- Each operator takes 0, 1, or 2 subexpressions
- Main operators:

Op.	Arity	Description	Condition
R	0	Relation name	$R \in \mathcal{L}$
$\rho_{A \to B}$	1	Renaming	$A, B \in \mathcal{L}$
$\Pi_{A_1 \ldots A_n}$	1	Projection	$A_1 \ldots A_n \in \mathcal{L}$
σ_ϕ	1	Selection	ϕ formula
$\times$	2	Cross product	
$\cup$	2	Union	
$\setminus$	2	Difference	
$\bowtie_\phi$	2	Join	ϕ formula

Relation name

Guest

id	name	email
1	John Smith	john.smith@gmail.com
2	Alice Black	alice@black.name
3	John Smith	john.smith@ens.fr

Reservation

id	guest	room	arrival	nights
1	1	504	2017-01-01	5
2	2	107	2017-01-10	3
3	3	302	2017-01-15	6
4	2	504	2017-01-15	2
5	2	107	2017-01-30	1

Expression: `Guest`

Result:

id	name	email
1	John Smith	john.smith@gmail.com
2	Alice Black	alice@black.name
3	John Smith	john.smith@ens.fr

Renaming

Guest

id	name	email
1	John Smith	john.smith@gmail.com
2	Alice Black	alice@black.name
3	John Smith	john.smith@ens.fr

Reservation

id	guest	room	arrival	nights
1	1	504	2017-01-01	5
2	2	107	2017-01-10	3
3	3	302	2017-01-15	6
4	2	504	2017-01-15	2
5	2	107	2017-01-30	1

Expression: $\rho_{\texttt{id}\to\texttt{guest}}(\texttt{Guest})$

Result:

guest	name	email
1	John Smith	john.smith@gmail.com
2	Alice Black	alice@black.name
3	John Smith	john.smith@ens.fr

Projection

Guest

id	name	email
1	John Smith	john.smith@gmail.com
2	Alice Black	alice@black.name
3	John Smith	john.smith@ens.fr

Reservation

id	guest	room	arrival	nights
1	1	504	2017-01-01	5
2	2	107	2017-01-10	3
3	3	302	2017-01-15	6
4	2	504	2017-01-15	2
5	2	107	2017-01-30	1

Expression: $\Pi_{\texttt{email},\texttt{id}}(\texttt{Guest})$

Result:

email	id
john.smith@gmail.com	1
alice@black.name	2
john.smith@ens.fr	3

Selection

Guest

id	name	email
1	John Smith	john.smith@gmail.com
2	Alice Black	alice@black.name
3	John Smith	john.smith@ens.fr

Reservation

id	guest	room	arrival	nights
1	1	504	2017-01-01	5
2	2	107	2017-01-10	3
3	3	302	2017-01-15	6
4	2	504	2017-01-15	2
5	2	107	2017-01-30	1

Expression: $\sigma_{\texttt{arrival}>\text{2017-01-12}\wedge\texttt{guest}=2}(\texttt{Reservation})$

Result:

id	guest	room	arrival	nights
4	2	504	2017-01-15	2
5	2	107	2017-01-30	1

The formula used in the selection can be any Boolean combination of comparisons of attributes to attributes or constants.

Cross product

Guest

id	name	email
1	John Smith	john.smith@gmail.com
2	Alice Black	alice@black.name
3	John Smith	john.smith@ens.fr

Reservation

id	guest	room	arrival	nights
1	1	504	2017-01-01	5
2	2	107	2017-01-10	3
3	3	302	2017-01-15	6
4	2	504	2017-01-15	2
5	2	107	2017-01-30	1

Expression: $\Pi_{\texttt{id}}(\texttt{Guest}) \times \Pi_{\texttt{name}}(\texttt{Guest})$

Result:

id	name
1	Alice Black
2	Alice Black
3	Alice Black
1	John Smith
2	John Smith
3	John Smith

Union

Guest

id	name	email
1	John Smith	john.smith@gmail.com
2	Alice Black	alice@black.name
3	John Smith	john.smith@ens.fr

Reservation

id	guest	room	arrival	nights
1	1	504	2017-01-01	5
2	2	107	2017-01-10	3
3	3	302	2017-01-15	6
4	2	504	2017-01-15	2
5	2	107	2017-01-30	1

Expression: $\Pi_{\texttt{room}}(\sigma_{\texttt{guest}=2}(\texttt{Reservation})) \cup \Pi_{\texttt{room}}(\sigma_{\texttt{arrival}=\text{2017-01-15}}(\texttt{Reservation}))$

Result:

room
107
302
504

Union

Guest

id	name	email
1	John Smith	john.smith@gmail.com
2	Alice Black	alice@black.name
3	John Smith	john.smith@ens.fr

Reservation

id	guest	room	arrival	nights
1	1	504	2017-01-01	5
2	2	107	2017-01-10	3
3	3	302	2017-01-15	6
4	2	504	2017-01-15	2
5	2	107	2017-01-30	1

Expression: $\Pi_{\texttt{room}}(\sigma_{\texttt{guest}=2}(\texttt{Reservation})) \cup \Pi_{\texttt{room}}(\sigma_{\texttt{arrival}=\text{2017-01-15}}(\texttt{Reservation}))$

Result:

room
107
302
504

This simple union could have been written $\Pi_{\texttt{room}}(\sigma_{\texttt{guest}=2 \vee \texttt{arrival}=\text{2017-01-15}}(\texttt{Reservation}))$. Not always possible.

Difference

Guest

id	name	email
1	John Smith	john.smith@gmail.com
2	Alice Black	alice@black.name
3	John Smith	john.smith@ens.fr

Reservation

id	guest	room	arrival	nights
1	1	504	2017-01-01	5
2	2	107	2017-01-10	3
3	3	302	2017-01-15	6
4	2	504	2017-01-15	2
5	2	107	2017-01-30	1

Expression: $\Pi_{\texttt{room}}(\sigma_{\texttt{guest}=2}(\texttt{Reservation})) \setminus \Pi_{\texttt{room}}(\sigma_{\texttt{arrival}=\text{2017-01-15}}(\texttt{Reservation}))$

Result:

room
107

Difference

Guest

id	name	email
1	John Smith	john.smith@gmail.com
2	Alice Black	alice@black.name
3	John Smith	john.smith@ens.fr

Reservation

id	guest	room	arrival	nights
1	1	504	2017-01-01	5
2	2	107	2017-01-10	3
3	3	302	2017-01-15	6
4	2	504	2017-01-15	2
5	2	107	2017-01-30	1

Expression: $\Pi_{\texttt{room}}(\sigma_{\texttt{guest}=2}(\texttt{Reservation})) \setminus \Pi_{\texttt{room}}(\sigma_{\texttt{arrival}=\text{2017-01-15}}(\texttt{Reservation}))$

Result:

room
107

This simple difference could have been written $\Pi_{\texttt{room}}(\sigma_{\texttt{guest}=2\wedge\texttt{arrival}\neq\text{2017-01-15}}(\texttt{Reservation}))$. Not always possible.

Join

Guest

id	name	email
1	John Smith	john.smith@gmail.com
2	Alice Black	alice@black.name
3	John Smith	john.smith@ens.fr

Reservation

id	guest	room	arrival	nights
1	1	504	2017-01-01	5
2	2	107	2017-01-10	3
3	3	302	2017-01-15	6
4	2	504	2017-01-15	2
5	2	107	2017-01-30	1

Expression: $\texttt{Reservation} \bowtie_{\texttt{guest}=\texttt{id}} \texttt{Guest}$

Result:

id	guest	room	arrival	nights	name	email
1	1	504	2017-01-01	5	John Smith	john.smith@gmail.com
2	2	107	2017-01-10	3	Alice Black	alice@black.name
3	3	302	2017-01-15	6	John Smith	john.smith@ens.fr
4	2	504	2017-01-15	2	Alice Black	alice@black.name
5	2	107	2017-01-30	1	Alice Black	alice@black.name

The formula used in the join can be any Boolean combination of comparisons of attributes of the table on the left to attributes of the table on the right.

Note on the join

- The join is not an elementary operator of the relational algebra (but it is very useful)
- It can be seen as a combination of renaming, cross product, selection, projection
- Thus:

$$\begin{aligned}&\texttt{Reservation} \bowtie_{\texttt{guest}=\texttt{id}} \texttt{Guest}\\ \equiv\ &\Pi_{\texttt{id},\texttt{guest},\texttt{room},\texttt{arrival},\texttt{nights},\texttt{name},\texttt{email}}(\\ &\quad\sigma_{\texttt{guest}=\texttt{temp}}(\texttt{Reservation} \times \rho_{\texttt{id}\to\texttt{temp}}(\texttt{Guest})))\end{aligned}$$

- If R and S have for attributes $\mathcal{A}$ and $\mathcal{B}$, we note $R \bowtie S$ the natural join of R and S, where the join formula is $\bigwedge_{A\in\mathcal{A}\cap\mathcal{B}} A = A$.

Bag semantics

In bag semantics (what is actually used by RDBMS):

- All operations return multisets
- In particular, projection and union can introduce multisets even when initial relations are sets

Extension: Aggregation

- Various extensions have been proposed to the relational algebra to add additional features
- In particular, aggregation and grouping [Klug, 1982, Libkin, 2003] of results
- With a syntax inspired from [Libkin, 2003]:

$$\sigma_{\texttt{avg}>3}(\gamma^{\texttt{avg}}_{\texttt{room}}[\lambda x.\texttt{avg}(x)](\Pi_{\texttt{room},\texttt{nights}}(\texttt{Reservation})))$$

computes the average number of nights per reservation for each room having an average greater than 3

room	avg
302	6
504	3.5

Plan

Relational calculus

- Logical language to express queries
- First-order logic formula, without function symbols, and with relation symbols the labels of the database schema (plus comparison predicates)
- Unnamed, untyped perspective
- Fix:
  - A set $\mathcal{X}$ of variables
  - A set $\mathcal{V}$ of values
  - A database schema S

Relational calculus: Syntax

- For every relation $R \in S$ of arity n, for every $(\alpha_1, \ldots, \alpha_n) \in (\mathcal{X} \cup \mathcal{V})^n$: $R(\alpha_1, \ldots, \alpha_n) \in \mathrm{FO}$
- Also allow equality predicate, possibly inequality
- For every $(\phi_1, \phi_2) \in \mathrm{FO}^2$, for every $x \in \mathcal{X}$:
  - $\phi_1 \land \phi_2 \in \mathrm{FO}$
  - $\phi_1 \lor \phi_2 \in \mathrm{FO}$
  - $\neg\phi_1 \in \mathrm{FO}$
  - $\forall x\ \phi_1 \in \mathrm{FO}$
  - $\exists x\ \phi_1 \in \mathrm{FO}$
- Free variables of $\phi \in \mathrm{FO}$: variables x appearing in ϕ and not qualified by a $\forall x$ or a $\exists x$
- One writes a relational calculus query in the form $Q(x_1, \ldots, x_m) = \phi$ where $x_1, \ldots, x_m$ are free variables of ϕ

Relational calculus: Semantics

- A relational calculus query on schema S can be seen as a function with input a database D over S and producing a relation as output
- adom(D): active domain of D, set of values in D
- If $Q(x_1, \dots, x_n) = \phi$ is a calculus query over S and D a database over S, then:

$$Q(D) = \{ (v_1, \dots, v_n) \in (\text{adom}(D))^n \mid D \models \phi[x_1/v_1, \dots, x_n/v_n] \}$$

where $D \models \phi$ is defined inductively:

- $D \models R(u_1, \dots, u_m) \iff R(u_1, \dots, u_m) \in D$
- $D \models \phi_1 \wedge \phi_2 \iff D \models \phi_1 \wedge D \models \phi_2$
- $D \models \phi_1 \vee \phi_2 \iff D \models \phi_1 \vee D \models \phi_2$
- $D \models \neg\phi_1 \iff D \not\models \phi_1$
- $D \models \forall x\ \phi_1 \iff \forall v \in \text{adom}(D)\ D \models \phi_1[x/v]$
- $D \models \exists x\ \phi_1 \iff \exists v \in \text{adom}(D)\ D \models \phi_1[x/v]$

Codd's theorem

Theorem ([Codd, 1972])

The relational algebra and the relational calculus are equivalent:

- *for every relational algebra query q over a schema S, there exists a relational calculus query Q over S such that for every database D over S, $q(D) = Q(D)$*
- *for every relational calculus query Q over a schema S, there exists a relational algebra query q over S such that for every database D over S, $q(D) = Q(D)$*

Furthermore, translating from one formalism to the other can be done in polynomial time.

Why is this important?

- Allows using a declarative formalism to specify queries: logics. . . or SQL
- These queries are then compiled via Codd's transformation into an algebraic formalism
- Algebraic queries are then optimized, by using the properties of the relational algebra (transformation rules, e.g., pushing selection within joins, exploiting associativity of joins, etc.)
- Optimized queries can then be evaluated, by exploiting the fact that each operator of the relational algebra can easily be implemented (in several different ways, to be chosen based on a cost function)
- This is RDBMS Implementation 101, a main reason of the success of RDBMSs!

Subclasses of queries

- Conjunctive query (CQ): relational calculus query without $\vee$, $\neg$, $\forall$
- Positive query (PQ): relational calculus query without $\neg$, $\forall$
- Union of conjunctive queries (UCQ): special case of positive query where the $\vee$ and $\wedge$ form a DNF formula

Subclasses of queries

- Conjunctive query (CQ): relational calculus query without $\vee$, $\neg$, $\forall$
- Positive query (PQ): relational calculus query without $\neg$, $\forall$
- Union of conjunctive queries (UCQ): special case of positive query where the $\vee$ and $\wedge$ form a DNF formula

Expressiveness

- CQs are equivalent to the relational algebra without $\cup$ and $\setminus$, and where σ does not feature disjunction
- UCQs are equivalent to PQs (but exponential blow-up), and equivalent to the relational algebra without $\setminus$

Plan

Motivation: recursive queries

- Query languages considered so far (relational algebra, calculus) have a limited horizon
- Some data structures (trees, graphs) require arbitrarily deep navigation, recursion
- How can we build a theory of recursive query languages?
- RDBMSs are not always adapted to this type of data/queries, cf. XML or graph DBMSs
- Example application: transitive closure of a graph $G(\mathit{from}, \mathit{to})$

Plan

Datalog

- Simplest recursive query language: adding recursion to conjunctive queries
- Inspired from logic programming
- Datalog query (or program): set of rules that produce intensional facts
- Schema of a Datalog program: classical database schema (extensional schema) + (disjoint) schema of intensional facts (intensional schema)
- Fix a distinguished relation *Goal* of the intensional schema, whose arity is the arity of the query

Syntax

Finite set of rules r of the form:

$$\underbrace{S(\mathbf{y})}_{\text{head}} \leftarrow \underbrace{R_1(\mathbf{x}_1), \ldots, R_n(\mathbf{x}_n)}_{\text{body}}$$

with:

- S relation of the intensional schema
- $R_1, \ldots, R_n$ relations of the intensional or extensional schemas
- $\mathbf{x}_1, \ldots, \mathbf{x}_n, \mathbf{y}$: tuples of variables (or possibly constants), of arity compatible with the relations
- Each variable in the head is present in the body

Fix-point semantics

- Each rule r of a program P can be seen as a conjunctive query on the database D:

$$r(D) := \{S(\mathbf{y}) \mid \exists z_1 \dots z_k \; R_1(\mathbf{x_1}) \in D \wedge \cdots \wedge R_n(\mathbf{x_n}) \in D\}$$

  where the z_i's are the variables of the rule body
- Consequence operator Γ_P defined by:

$$\Gamma_P(D) := D \cup \bigcup_{r \in P} \{r(D)\}$$

- We consider the sequence (D_n) defined by:
  $D_0 = D$, $D_{n+1} = \Gamma_P(D_n)$
- The semantics of P over D is the set of facts of the relation $Goal$ in D_∞, the fixpoint of the sequence (D_n)

Example: transitive closure

$$Goal(x, y) \leftarrow G(x, y)$$
$$Goal(x, y) \leftarrow Goal(x, z), G(z, y)$$

Plan

While operator

- Algebra: essentially imperative programming (in contrast to calculus)
- One can add to the algebra:
  - the possibility to define intermediary variables and to assign them values (does not affect expressive power)
  - a **while** operator of the form:

while change do

...assign value to one or several variables...

done

- Semantics: the content of the loop is evaluated while the assignments change the underlying variables
- Infinite loops possible!

Inflationary vs non-inflationary while

- Two variants:

  Non-inflationary: Assignment operator :=, arbitrary assignment

  Inflationary: Assignment operator +=, the assigned value can only grow
- Infinite loops impossible with an inflationary **while** (because the active domain is finite)
- Non-inflationary potentially more expressive

Example: transitive closure

We introduce a relation schema C with attributes *from* and *to*, similarly to G.

Non-inflationary

$C := G$

while change do

$C := C \cup \pi_{from,to}(\rho_{to\to int}(C) \bowtie \rho_{from\to int}(G))$

done

C

Inflationary

$C \mathrel{+}= G$

while change do

$C \mathrel{+}= \pi_{from,to}(\rho_{to\to int}(C) \bowtie \rho_{from\to int}(G))$

done

C

Plan

Non-inflationary fixpoint

- We add to the relational calculus a fixpoint construction
- Let $\phi(T)$ be a calculus formula, mentioning the schema relations as well as a new relation T, having for free variables $x_1, \ldots, x_n$
- Then $\mu_T[\phi(T)](x_1, \ldots, x_n)$ is a formula of the fixpoint calculus
- Semantics: in terms of the least fixpoint (if it exists) of the relation T, obtained by replacing T at each step with the set of facts of the form $T(x_1, \ldots, x_n)$ for $\phi(T)(x_1, \ldots, x_n)$ satisfied (starting with $T = \emptyset$)
- Equivalent to algebra with non-inflationary **while**!

Inflationary fixpoint

- We add to the relational calculus a fixpoint construction
- Let $\phi(T)$ be a calculus formula, mentioning the schema relations as well as a new relation T, having for free variables $x_1, \ldots, x_n$
- Then $\mu^+_T[\phi(T)](x_1, \ldots, x_n)$ is a formula of the fixpoint calculus
- Semantics: in terms of the least fixpoint of the relation T, obtained by adding to T at each step with the set of facts of the form $T(x_1, \ldots, x_n)$ for $\phi(T)(x_1, \ldots, x_n)$ satisfied (starting with $T = \emptyset$)
- Equivalent to algebra with inflationary **while**!

Example: transitive closure

Non-inflationary

$$\{\,(x,y)\mid \mu_C\,[G(x,y)\vee C(x,y)\vee(\exists z\,C(x,z)\wedge G(z,y))](x,y)\}$$

Inflationary

$$\{\,(x,y)\mid \mu^+_C\,[G(x,y)\vee(\exists z\,C(x,z)\wedge G(z,y))](x,y)\}$$

Plan

Query evaluation

- Query Q in some query language (CQ, FO, FO+μ, FO+μ^+...) – we will use a logical formalism here
- Database D (always finite!)
- Query evaluation: Computing $Q(D)$
- Complexity of this problem?
- To simplify the study of complexity, we often assume that Q is a Boolean query, i.e., it returns $\top$ or $\bot$

Data complexity

For some fixed Q, what is the complexity of computing $Q(D)$ in terms of the size of the database D?

Combined complexity

For some query language $\mathcal{Q}$, what is the complexity of computing $Q(D)$ in terms of the size of the query $Q \in \mathcal{Q}$ and of the database D?

Complexity classes

- We restrict to Boolean problems (returning $\top$ or $\bot$)
- Set of all problems solvable by a resource-constrained computing method:
- For example:

  PTIME: deterministic Turing machine in polynomial time

  NP: non-deterministic Turing machine in polynomial time

  PSPACE: deterministic Turing machine in polynomial space

  AC^0: Boolean circuit of polynomial size and constant depth
- We know that: $AC^0 \subsetneq \text{PTIME} \subseteq \text{NP} \subseteq \text{PSPACE}$
- Open whether $\text{PSPACE} \subseteq \text{PTIME}$ (!)

Membership and hardness for a class

- A problem P belongs to a complexity class $\mathcal{C}$ (or in $\mathcal{C}$) if it is solvable by the corresponding resource-constrained computing method
- A problem P is hard for a complexity class $\mathcal{C}$ (or $\mathcal{C}$-hard) if there exists a reduction that transforms whatever problem $P' \in \mathcal{C}$ into an instance of the problem P
- complete: in $\mathcal{C}$ + $\mathcal{C}$-hard
- Several ways to define reductions
- Here, we assume that there exists a function computable in polynomial time that transforms one instance I' of problem P' into an instance I of P such that $P(I) = P'(I')$

Descriptive complexity

- A query language $\mathcal{Q}$ captures a complexity class $\mathcal{C}$ if:
  - For all $Q \in \mathcal{Q}$, query evaluation of query Q in in $\mathcal{C}$ (data complexity)
  - For all problem P in $\mathcal{C}$, there exists a query $Q \in \mathcal{Q}$ such that evaluating Q exactly solves P (without a reduction)!
- If $\mathcal{Q}$ captures $\mathcal{C}$ and if $\mathcal{C}$ has problems that are complete for $\mathcal{C}$, then there exists $Q \in \mathcal{Q}$ such that Q is $\mathcal{C}$-complete, but the converse is not true

Plan

Data complexity

Theorem

CQ evaluation is PTIME in data complexity.

Proof.

By enumerating all valuations of variables of the query in the database. We will see later a much stronger result. □

Combined complexity

Theorem

CQ evaluation is NP-complete in combined complexity.

Proof.

Membership in NP is easy. Hardness for NP can be proved by reduction from graph 3-colorability. □

α-acyclic query

- A CQ can be seen as a hypergraph (vertices are variables, hyperedges the atoms of the CQ, labeled by the relation name)
- A hypergraph $\mathcal{H}$ has a join tree where one can find a tree whose nodes are labeled by the hyperedges of $\mathcal{H}$ and such that:
  - every hyperedge of $\mathcal{H}$ appears as the label of one node of the tree;
  - for every vertex x of $\mathcal{H}$, the set of tree nodes labeled by a hyperedge referring to x is a connected subtree
- A query is α-acyclic if its hypergraph has a join tree
- Can be obtained in linear time if it exists [Tarjan and Yannakakis, 1984]

Yannakakis's algorithm [Yannakakis, 1981]

- Algorithm to evaluate acyclic queries (non-necessarily Boolean):
  1. Construct the join tree
  2. Eliminate all useless tuples of a relation with the semijoin operator $\ltimes$: $R \ltimes S = \Pi_{R.*}(R \bowtie S)$ by navigating twice in the join tree: from bottom up, then from top down
  3. Evaluate the query bottom up, by computing joins following the tree and by projecting useless variables out as you go
- Polynomial complexity in the size of the query, the input, and the output (combined complexity)

Plan

Data complexity

Theorem

FO evaluation is PTIME in data complexity.

Proof.

By rewriting in prenex form and naive evaluation. □

FO does not capture the whole of PTIME

Theorem

One cannot compute in FO that a relation containing a total order has an even number of elements, or that a graph is connected.

Fairly complex to prove, relies on Ehrenfeucht–Fraïssé games (see [Libkin, 2004]).

Data complexity, more precise

Theorem

FO evaluation is AC^0 in data complexity.

Proof.

By rewriting to the relational algebra. □

Combined complexity

Theorem

FO evaluation is PSPACE-complete in combined complexity.

Proof.

Membership in PSPACE easy. Hardness for PSPACE from the QSAT problem. □

Plan

Data complexity

Theorem

Evaluation of FO+μ^+ is PTIME in data complexity.
Evaluation of FO+μ is PSPACE in data complexity.

Proof.

Direct. Only need polynomial space to detect infinite loops. □

Connectedness, parity

Theorem

Connectedness is expressible in FO+μ^+.

The problem of determining whether an arbitrary relation has an even number of elements (or tuples) cannot be expressed in FO+μ.

Proof.

First part easy, recall transitive closure query.
Second part shown in [Abiteboul et al., 1995], chapter 17. □

Parity and FO+μ^+, with order

Theorem

FO+μ^+ can compute if a relation that contains a total order has an even number of elements.

Proof.

Explicit construction. □

More general results

Theorem

$FO+\mu^+$ captures PTIME on databases including a total order relation of the domain.

$FO+\mu$ captures PSPACE on databases including a total order relation of the domain.

Proof.

Simulation of polynomial-time or polynomial-space deterministic Turing machines with a fixpoint computation. □

Data complexity (bis)

Theorem

Evaluating FO+μ^+ is PTIME-complete in data complexity.
Evaluating FO+μ is PSPACE-complete in data complexity.

Proof.

Hardness comes from descriptive complexity on ordered structures. □

Plan

Plan

Query optimization

- Goal: Given a query q in some query language $\mathcal{Q}$ and a database D, find a query equivalent to q on D and faster to evaluate on D
- Here: $\mathcal{Q}$ in the relational calculus (or a fragment thereof), and one looks for a query faster on whatever database (we do not look D, we perform static analysis)
- In actual RDBMSs: $\mathcal{Q}$ is the set of query execution plans (a specialization of the relational algebra where implementations are chosen for each operator) and statistics on D are used

Global optimization

- We consider global optimization techniques, considering a query in its entirety (techniques on execution plans are more local, e.g., local rewritings)
- We formally define:

  Equivalence: $q \equiv q'$ if for all database D, $q(D) = q'(D)$

  Minimality: q' is the "best" query equivalent to q in $\mathcal{Q}$

Containment and equivalence

Definition

A query q is contained in a query q' (denoted $q \sqsubseteq q'$) if for all database D, $q(D) \subseteq q'(D)$

Containment and equivalence

Definition

A query q is contained in a query q' (denoted $q \sqsubseteq q'$) if for all database D, $q(D) \subseteq q'(D)$

Proposition

$q \equiv q'$ *iff* $q \sqsubseteq q'$ *and* $q' \sqsubseteq q$.

Proof.

Immediate. □

Plan

Case of the conjunctive queries

- We consider conjunctive queries (CQ) of the form:

$$q(\mathbf{x}) \leftarrow \exists \mathbf{y} : R_1(\mathbf{z}_1) \wedge \cdots \wedge R_n(\mathbf{z}_n)$$

  where each $\mathbf{z}_i$ is a tuple of variables among $\mathbf{x}$ and $\mathbf{y}$, and where each x_j appears at least in one $\mathbf{z}_j$
- Set semantics: for all database D, $q(D)$ is a finite set of tuples

Homomorphism

Definition

A homomorphism from a CQ q to a CQ q' is a function ϕ from the variables $\mathbf{x}, \mathbf{y}$ of q fo the variables $\mathbf{x}', \mathbf{y}'$ of q' such that:

- $\phi(\mathbf{x}) = \mathbf{x}'$
- for every atom $R(\mathbf{z}_i)$ of q, there exists an atom $R(\mathbf{z}'_{i'})$ of q' such that $\phi(\mathbf{z}_i) = \mathbf{z}'_{i'}$

Definition

A homomorphism is an isomorphism if it is one-to-one and its converse is a homomorphism.

Instance associated to a query

Definition

For all conjunctive query

$$q(\mathbf{x}) \leftarrow \exists \mathbf{y} : R_1(\mathbf{z}_1) \wedge \cdots \wedge R_n(\mathbf{z}_n)$$

one can construct the instance associated to q, denoted I_q, where the active domain is $\{ a_z \mid z \in \mathbf{x} \cup \mathbf{y} \}$ and which is formed of the n tuples $R(a_{z_{i1},\ldots,z_{ik}})$ for $R(z_{i1}, \ldots, z_{ik})$ atom of q

Instance associated to a query

Definition

For all conjunctive query

$$q(\mathbf{x}) \leftarrow \exists \mathbf{y} : R_1(\mathbf{z}_1) \wedge \cdots \wedge R_n(\mathbf{z}_n)$$

one can construct the instance associated to q, denoted I_q, where the active domain is $\{ a_z \mid z \in \mathbf{x} \cup \mathbf{y} \}$ and which is formed of the n tuples $R(a_{z_{i1},\dots,z_{ik}})$ for $R(z_{i1}, \dots, z_{ik})$ atom of q

Proposition

For all CQs $q(\mathbf{x})$, $q'(\mathbf{x}')$, *there exists a homomorphism from* q *to* q' *iff* $(a_{x'_1}, \dots, a_{x'_j}) \in q(I_{q'})$.

Homomorphism theorem

Theorem ([Chandra and Merlin, 1977])

For all CQs q, q', $q \sqsubseteq q'$ iff there exists a homomorphism from q' to q.

Minimal query

Definition

A conjunctive query is minimal if it has a minimal number of atoms among all equivalent conjunctive queries.

Minimal query

Definition

A conjunctive query is minimal if it has a minimal number of atoms among all equivalent conjunctive queries.

- Translation of a CQ to an algebra query: if there are n atoms, we obtain $n - 1$ joins
- Joins are the most costly operations of the relational algebra (bar cross products)
- Finding a minimal query amounts to global optimization

Unicity of minimal query

Proposition ([Chandra and Merlin, 1977])

Let q be a CQ. Then there exists a CQ q' obtained by removing atoms from q which is minimal.

Proof.

Consider a minimal query equivalent to q and apply the homomorphism theorem. □

Proposition ([Chandra and Merlin, 1977])

Let q, q' be two equivalent minimal CQs. Then there exists an isomorphism from q to q'.

Proof.

Apply the homomorphism theorem. The image by the homomorphism is an equivalent minimal query. □

Minimization algorithm

Apply the following procedure to minimize a query:

> *For every atom of the query, test if there exists an equivalent query not containing this atom, and thus if there exists a homomorphism sending this atom to another atom of the query. If so, delete it, and continue until obtaining an equivalent minimal query.*

Complexity issues

Proposition

The following problems are NP-complete:

- *given two CQs q, q', determine whether $q \sqsubseteq q'$*
- *given two CQs q, q', determine whether $q \equiv q'$*
- *given a CQ q, determine if q is non-minimal*

Proof.

NP-hardness is by reduction from 3-colorability, as for combined complexity of query evaluation. Membership in NP is direct. □

Complexity issues

Proposition

The following problems are NP-complete:

- *given two CQs q, q', determine whether $q \sqsubseteq q'$*
- *given two CQs q, q', determine whether $q \equiv q'$*
- *given a CQ q, determine if q is non-minimal*

Proof.

NP-hardness is by reduction from 3-colorability, as for combined complexity of query evaluation. Membership in NP is direct. □

NP-hard... in the queries. Queries may be small enough so that an exponential algorithm may not be an issue.

Bag semantics [Chaudhuri and Vardi, 1993]

- In practice, RDBMSs implement a bag semantics
- Two queries in bag semantics are equivalent if and only if they are isomorphic (intuitively, because two similar but non isomorphic queries can introduce a different number of results)
- Query containment: Π_2^P-hard claimed (but not proved). Decidability (and precise complexity if decidable): open!

Plan

Satisfiability in the relational calculus

Definition

A Boolean relational calcululs query q is satisfiable if there exists a (finite) database D such that $D \models q$.

Satisfiability in the relational calculus

Definition

A Boolean relational calcululs query q is satisfiable if there exists a (finite) database D such that $D \models q$.

Theorem ([Trakhtenbrot, 1963])

Satisfiability of the relational calculus (in the finite case) is undecidable.

Proof.

Reduction possible from the POST correspondence problem, technical, see [Abiteboul et al., 1995]. □

Containment and equivalence of the calculus

Theorem

Containment and equivalence of relational calculus queries are undecidable and co-recursively enumerable.

Containment and equivalence of the calculus

Theorem

Containment and equivalence of relational calculus queries are undecidable and co-recursively enumerable.

Proof.

Undecidability is by direct reduction from the undecidability of satisfiability.

Co-recursive enumerability is shown directly, by enumerating possible counter-examples. □

Plan

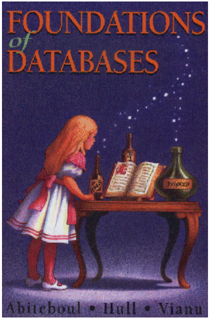
Plan

Database theory: a rich field of research

- Rich connections with various areas of theoretical computer science and mathematics: logics, algorithms, graph theory, automata theory, typing theory, algebra, etc.
- This lecture: very high-level overview, no time for proofs, some quite interesting
- Further lectures at EPIT: more on efficient query evaluation, XML and graph databases, descriptive complexity, logical independence, uncertainty in databases, connections with description logics
- Plenty of other interesting topics: data exchange and integration, probabilistic databases, theory of indexing structures, ranking and top-k queries…

Reference

- Main database theory textbook: [Abiteboul et al., 1995]



- In this lecture: (part of) chapters 3, 4, 5, 6, 12, 14, 17

Bibliographie I

Serge Abiteboul, Richard Hull, and Victor Vianu. *Foundations of Databases*. Addison-Wesley, 1995.

Ashok K. Chandra and Philip M. Merlin. Optimal implementation of conjunctive queries in relational data bases. In *Proceedings of the 9th Annual ACM Symposium on Theory of Computing, May 4-6, 1977, Boulder, Colorado, USA*, pages 77–90, 1977. doi: 10.1145/800105.803397. URL `http://doi.acm.org/10.1145/800105.803397`.

Surajit Chaudhuri and Moshe Y. Vardi. Optimization of *Real* conjunctive queries. In *Proceedings of the Twelfth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, May 25-28, 1993, Washington, DC, USA*, pages 59–70, 1993. doi: 10.1145/153850.153856. URL `http://doi.acm.org/10.1145/153850.153856`.

Bibliographie II

Edgar F. Codd. Relational completeness of data base sublanguages. In *Data Base Systems. Courant Computer Science Symposium*, 1972.

Anthony C. Klug. Equivalence of relational algebra and relational calculus query languages having aggregate functions. *J. ACM*, 29(3):699–717, 1982.

Leonid Libkin. Expressive power of SQL. *Theor. Comput. Sci.*, 296(3):379–404, 2003.

Leonid Libkin. *Elements of Finite Model Theory*. Texts in Theoretical Computer Science. An EATCS Series. Springer, 2004. doi: 10.1007/978-3-662-07003-1. URL `http://dx.doi.org/10.1007/978-3-662-07003-1`.

Bibliographie III

Robert Endre Tarjan and Mihalis Yannakakis. Simple linear-time algorithms to test chordality of graphs, test acyclicity of hypergraphs, and selectively reduce acyclic hypergraphs. *SIAM J. Comput.*, 13(3):566–579, 1984. doi: 10.1137/0213035. URL `http://dx.doi.org/10.1137/0213035`.

Boris A. Trakhtenbrot. Impossibility of an algorithm for the decision problem in finite classes. *American Mathematical Society Translations Series 2*, 23:1–5, 1963.

Mihalis Yannakakis. Algorithms for acyclic database schemes. In *VLDB*, pages 82–94, 1981.