Simplification of Schema Tuple Relational Queries

Hans Olofsson dva98hon@cs.umu.se

11th October 2004

Abstract

The $\mathcal{Q}$-language has several beneficial properties, among them are closure under syntactic query difference and decidability for satisfiability. But in calculating query intersections and differences, the resulting queries can quickly become hard to read even if they state something simple. Thus it would be desirable to be able to reduce queries to unique forms that are as concise as possible. As the $\mathcal{Q}$-language is defined by the closure of the $\mathcal{L}$-language over disjunction, the first steps toward simplification of queries can be done through simplification of $\mathcal{L}$-queries. This thesis investigated the possibility of a minimum/canonical form for $\mathcal{L}$-queries and more importantly developed an algorithm which reduces the queries to a 'unique' form with little or no redundancy in the $\mathcal{L}$-language.

Contents

Chapter 1

Introduction

One of the most common models used today in databases is the relational model. This is generally ascribed to the fact that it has a strong foundation in logic[3]. Furthermore it is possible to ask questions in tuple relational calculus over that model. Since Q (and by extension $\mathcal{L}$) are limited fragments of the tuple calculus, they can also be used to generate queries over databases (of the relational model). This is good because the things that give $\mathcal{L}$ and Q their limitations also gives them extremely nice properties (determinable for satisfiability, equality and subsumption)[4].

Now this thesis deals with finding a minimal canonical form of queries in $\mathcal{L}$, where minimal implies the smallest according to a measure and canonical refers here to a unique representative of the minimal group. Desired properties of the measure would be that it reduces redundancies in the query and possible evaluation time.

One of the reasons for checking a query for satisfiability, is that when asking queries over distributed databases it may be beneficial to ensure that the query is satisfiable. This is because it may not be obvious from the result until we have gathered the results of all the subqueries over the different databases that it has become unsatisfiable. Furthermore on distributed databases we definitely would benefit if the query used as few variables as possible.

Another reason is that one can in $\mathcal{L}$ decide if a query subsumes another query, by generating a couple of queries and checking theirs satisfiability. This could be used as in [5] to translate queries into natural language. Unfortunately one then need to process the query to gather information on relevant constants. Such cases there would certainly become easier if one could write[1] the subsuming query into the subsumed one and thus knowing which variable one should retrieve the restrictions from. There are also problems where the restrictions may be derived through other joins and while having a minimal and canonical form does not really alleviate these problems, the methods of achieving a minimal and canonical form from a query certainly does. This is because if you could write a query into another you could also expand[2] it to be equivalent with the other query and there is also a need to able to determine what the derived restrictions are on each attribute to change/remove that attributes conditions. The assertion that one actually can write a subsuming query into the subsumed query will not be proven here, but it will at least hold for those sub languages that are shown here to have a unique minimal canonical form.

[1] By matching variables

[2] By adding new variables and conditions

Chapter 2

Definitions

This section is mainly taken from [4], with some addendum on more restrictive languages than $\mathcal{L}$.

2.1 Foundations

We assume the existence of two disjoint, countable sets: $\mathcal{U}$, the *universal domain* of *atomic values*, and $\mathcal{P}$, *predicate names*. Let U be a distinct symbol representing the type of $\mathcal{U}$. A *relation schema* R is an n-tuple $[U,\ldots,U]$ where $n \geq 1$ is called the *arity* of R. A *database schema* D is a sequence $\langle P_1 : R_1,\ldots,P_m : R_m\rangle$, where $m \geq 1$, P_i's are distinct predicate names and R_i's are relation schemas. A *relation instance* r of R with arity n is a finite subset of $\mathcal{U}^n$. A *database instance* d of D is a sequence $\langle P_1 : r_1,\ldots,P_m : r_m\rangle$, where r_i is an instance of R_i for $i \in [1\ldots m]$.

Definition 2.1 *(Schema tuples)*
A *schema tuple* τ of the database instance d is the pair $\langle P_i : \mu\rangle$, where $1 \leq i \leq m$ and $\mu \in r_i$.

We say that the *type* of τ is P_i and we say the components of τ are the components of μ. The schema tuple τ_1 is equal to the schema tuple τ_2 if and only if τ_1 and τ_2 match on type and they match on all components. Thus if $\tau_1 = \langle$IsDirector : ['0133093','43252']$\rangle$ and $\tau_2 = \langle$ISCastMember: ['0133093','43252']$\rangle$ then $t_1 \neq t_2$. The positional access operator is extended to the schema tuples to mirror the standard tuple relational calculus. Thus $\tau_1[2] =$ '43252'. Furthermore we shall assume that tuple components may be accessed through attribute names (e.g. $\tau_1.entertainerID$). Finally we shall assume that $\mathcal{U}$ is totally ordered so that arithmetic comparison operators ($=,>,<,\geq,\leq$ and $\neq$) are well defined.

We now recursively define the set of *tuple relational formulas*. *Atomic formulas* provide the base for the inductive definition. The atomic formula $P(z)$, where P is a predicate name and z is a tuple variable, means that the tuple referred to by z is a schema tuple of type P. We term such formulas *range conditions*. $X\theta Y$ is an atomic formula where X and Y are either constants or component references (of the form $z.a$) and θ is one of the arithmetic comparison operators. We term such formulas to be *simple conditions* if either X or Y are constants and to be *join conditions* if both X and Y are component references. Lastly we include atomic formula $X\varepsilon C$ where X is a component reference, C is a set of constants and ε is a set membership operator ($\in$ and $\notin$).We term

such formulas *set conditions*. Finally if F_1 and F_2 are tuple relational formula, where F_1 has some free variable z, then $F_1 \wedge F_2$, $F_1 \vee F_2$, $\neg F_1$, $(\exists z)F_1$ and $(\forall z)F_1$ are also tuple relational formulas.

We now define the notion of a schema tuple query.

Definition 2.2 *(Schema tuple queries)*
A *schema tuple query* is an expression of the form $\{x|\varphi\}$, where φ is a tuple relational formula over the single free tuple variable x.

When we write the expression $\{x|\varphi\}$ we normally assume that the expression φ is over the free variable x. For the schema tuple τ, $\tau \in \{x|\varphi\}$ iff $\{\tau/x\}(\varphi)$ where $\{t/x\}\varphi$ means to substitute the term t in place of x in φ. Thus a *schema tuple query* is simply the intensional description of a schema tuple set. The actual tuples within this set are the extensional answers to the query.

We shall now turn our attention to several interesting sub-languages of the tuple relational formula that may be used to specify safe schema tuple queries.

2.2 The schema tuple query language $\mathcal{L}$

A *basic query component* built over the free tuple variable x is a formula of the form $(\exists y_1)(\exists y_2)\ldots(\exists y_n)\Psi$, where Ψ is a conjunction of range conditions, simple conditions, set conditions and join conditions using exactly the tuple variables $x,y_1,\ldots,y_n$. A *signed basic query component* is either a basic query component or the formula $\neg\Theta$ where Θ is a basic query component.

Definition 2.3 *(The language $\mathcal{L}$)*
The language $\mathcal{L}$ consists of all formulas of the form:

$$P(x) \wedge (\bigwedge_{i=1}^{k} \Phi_i)$$

where $P(x)$ restricts the free tuple variable x to range over P and Φ_i is a signed basic query component over x.

Definition 2.4 *(The language $\mathcal{L}_{pos}$)*
The language $\mathcal{L}_{pos}$ consists of all formulas of the form:

$$P(x) \wedge (\bigwedge_{i=1}^{k} \Theta_i)$$

where $P(x)$ restricts the free tuple variable x to range over P and Θ_i is a basic query component over x.

Definition 2.5 *(The language $\mathcal{L}_{simple}$)*
The language $\mathcal{L}_{simple}$ consists of all formulas of the form:

$$P(x) \wedge \Theta$$

where $P(x)$ restricts the free tuple variable x to range over P and Θ is a basic query component without range and join conditions.

Chapter 3

Minimality and canonical forms

While we want to find a canonical definition, I will begin with defining a minimal form and follow it up with a sorted form. The canonical form will then be loosely from those two definitions.

At first it might seem like a good idea to define a minimal query by merely counting its conditions, but this disregards several factors.

1. We do not want a set with only one member in the minimal query.
2. We do not want a set with irrelevant members in the minimal query.
3. We do not want to use a join condition instead of simple conditions.

This can of course be achieved by counting the set's length in a set conditions as something more than any other condition and weighting join conditions as heavier than simple conditions.

Definition 3.1 *(Length on $\mathcal{L}$)*
Let f be a function $\mathcal{L} \to \mathbb{N}$ which is defined by: $f(l) = \sum_{c \in l} h(c)$ where $h(c)$ is defined by

$$h(c) = \begin{cases} 3 & c = \textit{simple condition} \\ 3 & c = \textit{range condition} \\ 4 & c = \textit{join condition not of } \neq \textit{ type} \\ 5 & c = \textit{join condition of } \neq \textit{ type} \\ 3+n & c = \textit{set condition, and n is the length of the set} \\ \sum_{s \in c} h(s) & c = \textit{component} \end{cases}$$

The main reason to that a simple condition cost 3 has to do with replacing groups of simple conditions with set conditions.

Definition 3.2 *(Minimality of $\mathcal{L}$)*
A query $l \in \mathcal{L}$ is minimal if there exists no query $s \in \mathcal{L}$ where $l \equiv s$ and $f(s) < f(l)$.

Definition 3.3 *(Enumeration of variables)*
ϕ' is an enumeration of the variables in a query component if it is a 1-1 mapping from the variables in the component to $\{1 \ldots n\}$, where n is the number of variables in the component. A enumeration ϕ of the variables in a $\mathcal{L}$-query is a group of ϕ' enumerations, one for each of the query's components.

Let $>_s$ and $\leq_s$ be a lexical comparison operator (with the order $=, \leq, \geq, <, >, \neq, \in, \notin, 0-9$,A-Za-z) and $\phi(c)$ is replacing the variables in c by their enumeration in ϕ

Definition 3.4 *(Sorted form on $\mathcal{L}$)*
A query $l \in \mathcal{L}$ is in sorted form given enumeration ϕ of the variables if for each component:

1. There exists no join condition before a range condition.
2. There exists no simple condition before a range condition.
3. There exists no set condition before a range condition.
4. There exists no simple condition before a join condition.
5. There exists no set condition before a join condition.
6. There exists no set condition before a simple condition.
7. There exists no join condition $c : X \theta Y$ where $\phi(X) >_s \phi(Y)$.
8. There exists no two conditions c_1 and c_2 where c_1 is the join condition $Y = Z$ (with $\phi(Z) >_s \phi(Y)$) and c_2 is a join, set or simple condition over Z.
9. There exists no $c_1 \wedge c_2$ where they both are range conditions and $\phi(c_1) >_s \phi(c_2)$.
10. There exists no $c_1 \wedge c_2$ where they both are join conditions and $\phi(c_1) >_s \phi(c_2)$.
11. There exists no $c_1 \wedge c_2$ where they both are simple conditions and $\phi(c_1) >_s \phi(c_2)$.
12. There exists no $c_1 \wedge c_2$ where they both are set conditions and if the sets are written are strictly increasing vectors in $\mathcal{U}$ then $\phi(c_1) >_s \phi(c_2)$.

Definition 3.5 *(Canonical form on $\mathcal{L}$)*
$l \in \mathcal{L}$ is in canonical form if every component has a distinct enumeration and has been sorted by it.

3.1 Minimality on $\mathcal{L}$

Lemma 3.1 *(Minimal form in $\mathcal{L}$)*
Let $L \subset \mathcal{L}$ be all queries that are equivalent to $l \in \mathcal{L}$, then there exists a $l_m \in L$ such that $f(l_m) = \inf{}^1{}_{s \in L} f(s)$. This l_m is said to be a minimal form of l in $\mathcal{L}$

Proof 3.1 *(Of above)*
Since $f : \mathcal{L} \to \mathbb{N}$ we know that $\{f(s) : s \in L\}$ is non-empty (contains $f(l)$) and a subset of $\mathbb{N}$, which has the property of a least member[2]. And such a member would by definition fulfill the minimality definition given earlier. □

[1] $\inf S$ is the greatest lower bound of S

Chapter 4

Minimal canonical form of $\mathcal{L}_{simple}$

The first thing to realize is that we need no enumeration of the variables here, this is because we only have one variable. So basically just sorting any query puts it in canonical form. What now needs to be proven is that the minimal canonical form is unique.

Lemma 4.1 *(Simple condition uniqueness)*
A $l \in \mathcal{L}_{simple}$ with only one simple condition is in a unique minimal form.

Proof 4.1 *(Simple condition uniqueness)*
The simple conditions are arithmetic comparison operations on $\mathcal{U}$ and $\mathcal{U}$ is totally ordered, so we can see the simple conditions as subsets of $\mathcal{U}$. If $c : X\theta Y$ then the subset C would be:

- $\theta := $ gives $C = \{x | x \in \mathcal{U} \wedge x = Y\}$
- $\theta :< $ gives $C = \{x | x \in \mathcal{U} \wedge x < Y\}$
- $\theta :\leq $ gives $C = \{x | x \in \mathcal{U} \wedge x \leq Y\}$
- $\theta :> $ gives $C = \{x | x \in \mathcal{U} \wedge x > Y\}$
- $\theta :\geq $ gives $C = \{x | x \in \mathcal{U} \wedge x \geq Y\}$
- $\theta :\neq $ gives $C = \{x | x \in \mathcal{U} \wedge x \neq Y\}$

Furthermore we can assert that $\mathcal{U} = \mathbb{Q}$ (both are totally ordered and countable[2]). Then both $=$ and $\neq$ are unique (they have finite sets or finite complements, while the others have both set and complement infinite). We now have $C = \{x | x \in \mathcal{U} \wedge x\theta_1 Y_1\} = \{x | x \in \mathcal{U} \wedge x\theta_2 Y_2\}$ where $\theta_1, \theta_2 :<, \leq, >, \geq$. If we assume that $\theta_1 :<, \leq$ and $\theta_2 :>, \geq$ then there is a x which is less than both Y_1 and Y_2 thus making it a member of C and not a member of C at the same time (a contradiction). Therefore we can assert that $\theta_1, \theta_2 :<, \leq$ (or $>, \geq$). Also if θ_1 would be the same as θ_2 then $Y_1 \neq Y_2$ (say $Y_1 < Y_2$) but then there exists a Y_3 between Y_1 and Y_2 (a property of $\mathbb{Q}$) which is in C and not in C. Lastly we have that θ_1 and θ_2 are different (say $<$ respectively $\leq$), but then $Y_1 \neq Y_2$ (otherwise Y_1 is in and not in C) and again there would be a Y_3 between them which would be in and not in C. $\square$

Lemma 4.2 *(Set condition have a minimal canonical form)*
A $l \in \mathcal{L}_{simple}$ with only one set condition has a minimal canonical form.

Proof 4.2 *(Set condition have canonical form)*
The set conditions $c : X\varepsilon\{Y_1,\ldots,Y_n\}$ ($n \in \mathbb{N}$) can be rewritten as subsets of $\mathcal{U}$:

- $\varepsilon :\in$ gives $C = \{Y_1,\ldots,Y_n\}$

- $\varepsilon :\notin$ gives $C = \mathcal{U}\backslash\{Y_1,\ldots,Y_n\}$

Furthermore we are going to assume for simplicity that $Y_i \neq Y_j$ for $i \neq j$ and $Y_i \in \mathcal{U}$. Let $c_1 : X\varepsilon_1\{A_1,\ldots,A_m\}$ and $c_2 : X\varepsilon_2\{B_1,\ldots,B_l\}$. If we as previously assert that $\mathcal{U} = \mathbb{Q}$ then ε_1 and ε_2 must be the same operator and from the definition of C we note that $\{Y_1,\ldots Y_n\} = \{A_1,\ldots,A_m\} = \{B_1,\ldots,B_l\}$ which makes them the same if one removes the redundant set members of $\{A_1,\ldots,A_m\}$ and $\{B_1,\ldots,B_l\}$. Furthermore the simple conditions that can challenge them are $\theta :=, \neq$ (because of the finite set or finite complement in $\mathbb{Q}$). But if $n > 1$ the set (or the complements) size would be n which is impossible for those simple conditions. Although when $n = 1$ there exists simple condition describe the set condition and since a simple condition is has less weight in f than a set condition, it would then be the minimal form and unique by lemma 4.1. □

Lemma 4.3 *(Two simple conditions have a minimal canonical form)*
A $l \in \mathcal{L}_{simple}$ with only two simple condition has a canonical minimal form which is unique.

Proof 4.3 *Lemma 4.3*
The simple conditions are arithmetic comparison operations on $\mathcal{U}$ and $\mathcal{U}$ is totally ordered, so we can se simple conditions as subsets of $\mathcal{U}$. Using the C set definition from proof 4.1 we can define the two conditions as two sets C_1, C_2. And if they where over the same attribute we will have one of these three cases:

$C_1 \subseteq C_2 \vee C_2 \subseteq C_1$**:** In this case (assume $C_1 \subseteq C_2$) we only have one condition describing the query and from lemma 4.1 we know that it would be unique.

$C_1 \cap C_2 = \emptyset$**:** Here the query is false, thus have an unique minimal form.

$C_1 \cap C_2 = C_3$**:** In this case we have $C_3 = \{x | x \in \mathcal{U} \wedge x\theta_1 Y_1 \wedge x\theta_2 Y_2\}$ and we know that neither of θ_1 or θ_2 is $=$, since then it would be in either of the earlier cases. If θ_1 and θ_2 both were either less or greater comparison operators then either C_1 or C_2 would be C_3, so we can say that $\theta_1 :<, \leq, \neq$ and $\theta_2 :>, \geq, \neq$. If the operators are $\leq, \geq$ with $Y_1 = Y_2$ then $C_3 = \{Y_1\}$ (which is uniquely described) or if the operators are $\leq, \neq$ or $\neq, \geq$ with $Y_1 = Y_2$ then C_3 can be described by using the strict comparison operator. In the case where the operators are $\neq, \neq$ we describe the set with an set condition $c : X \notin \{Y_1, Y_2\}$, and from lemma 4.2 has a canonical form. In the other cases we just keep the original conditions, and to see the uniqueness we again assert $\mathcal{U} = \mathbb{Q}$ and note that C_3 is either a bounded interval (on $\mathbb{Q}$) or an unbounded interval intersected with the complement to the point. Using that in $\mathbb{Q}$ an interval has an infinite amount of points (or one point but that was taken care of above) and an infinite complement we know that we can not use set conditions to describe it. Furthermore we can easily see that it takes more than one simple condition to describe it, so the canonical form is the sorting of those two conditions.

If they were not over the same attribute then neither of them can be removed and sorting the query gives the unique form. □

Lemma 4.4 *(Two set conditions have a minimal canonical form)*
A $l \in \mathcal{L}_{simple}$ consisting of two set condition has a minimal form which is unique.

Proof 4.4 *(Two set conditions have a minimal canonical form)*
If the set conditions are over different variables or attributes then each of the set conditions can be rewritten in a unique minimal form and together they form a minimal form, which by sorting becomes unique. Let $c_1 : X\varepsilon_1\{A_1,\ldots,A_m\}$ and $c_2 : X\varepsilon_2\{B_1,\ldots,B_l\}$ be the set conditions and C_1, C_2 be their set representatives as defined in the proof 4.2. Since $C = C_1 \cap C_2$ we can divide into three cases:

- $\varepsilon_1, \varepsilon_2$ are $\in$ then $C = \{A_1,\ldots,A_m\} \cap \{B_1,\ldots,B_l\} = \{Y_1,\ldots,Y_n\}$ where $Y_i = A_j = B_k$ $0 < i \leq n$. And since $n \leq \min(l,m)$ C is finite it can be described as a set condition or false if $n = 0$.

- $\varepsilon_1, \varepsilon_2$ are $\notin$ then $C = \mathcal{U}\backslash\{A_1,\ldots,A_m\} \cap \mathcal{U}\backslash\{B_1,\ldots,B_l\} = \mathcal{U}\backslash(\{A_1,\ldots,A_m\} \cup \{B_1,\ldots,B_l\})$. Therefore $\{Y_1,\ldots,Y_n\} = \{A_1,\ldots,A_m\} \cup \{B_1,\ldots,B_l\}$ and $\max(m,l) \leq n \leq m+l$ is finite and $n > 0$ so it can be described as a set condition.

- $\varepsilon_1, \varepsilon_2$ are one of each $\in$ and $\notin$ (say $\varepsilon_1 : \in$ and $\varepsilon_2 : \notin$). Then we can write C as $\{A_1,\ldots,A_m\} \cap (\mathcal{U}\backslash\{B_1,\ldots,B_l\} = \{A_1,\ldots,A_m\}\backslash\{B_1,\ldots,B_l\} = \{Y_1,\ldots,Y_n\}$ which then would be a finite set and thus describable as a set condition or if $n = 0$ inherently $false$.

□

Lemma 4.5 *(A set and a simple conditions have a minimal canonical form)*
A $l \in \mathcal{L}_{simple}$ with only a set and a simple condition has a canonical minimal form which is unique.

Proof 4.5 *(A set and a simple conditions have a minimal canonical form)*
First we are going to assume that they both are over the same variable and attribute (otherwise the each have a unique minimal form). From earlier we know we can write the conditions as subsets of $\mathcal{U}$. And then we have $C = C_1 \cap C_2$, where C_1 and C_2 are the sets representing each condition. Let C_1 be the subset representing the set condition. If we look at C we can divide it into three cases (which of two is trivial).

C **finite:** Then C can be described as a set condition (or $false$)

C**'s complement is finite:** Then C can be described as a set condition (or $true$).

C **and its complement is infinite:** Then we know that C_1 was a $\notin$, and C_2 was not $=$ or $\neq$. So basically $C = \{x | x \in \mathcal{U} \wedge x\theta Y \wedge x \neq Y_1 \wedge \ldots \wedge x \neq Y_n\}$ where $\theta : <, \leq, >, \geq$. First we can remove the Y_i which does not uphold $Y_i\theta Y$ $(1 \leq i \leq n)$, lets call the remaining Y_i for $A_1,\ldots,A_m$ $(m \leq n)$. Furthermore if $A_j = Y$ for some $1 \leq j \leq m$ then we can remove A_j and change θ to θ', the strict version of θ. Now C consists of the intersection of an interval and the complement of a finite set of internal points (of the interval). We can assume that the finite set is non-empty (if it is empty then we only have a simple condition describing C so it is a unique minimal form). And since in $\mathbb{Q}$ no finite sequence of intersection of complements

to points becomes an interval, we know that the interval must be a condition and then the complement of a finite set of points becomes the other condition in its minimal form.

□

Theorem 4.6 *(Minimal canonical form in $\mathcal{L}_{simple}$)*
Let $l \in \mathcal{L}_{simple}$, then there exists a unique $l_m \in \mathcal{L}_{simple}$ such that l_m is a minimal canonical form

Before we continue to the proof there are some observation that should be done. To realize the minimal maxima of simple and set conditions on an attribute we divide the conditions into five types. These types are less $(<, \leq)$, greater $(>, \geq)$, equal, not equal and set $(\in, \notin)$. Now from lemma 4.4 and lemma 4.3 it follows that if we have two conditions of the same type, then we can reduce it into one condition and if we have an equal condition then the query is either false or we can remove the rest of the conditions. This means that we may now have at most for conditions one less, greater, not equal and a set. Now from lemma 4.5 we realize that a not equal and a set condition is reducible into one condition. This means we can at most have three conditions and two examples of when in occur are $X < 5$,$X > 2$,$X \neq 3$ and $X < 6$,$X > 2$,$X \notin \{3,4\}$. Also from the same lemma we know that if we have a $\in$-set condition then the group can be reduced to one condition.

Proof 4.6 *Proof of theorem 4.6*
We are going to use induction over the number of attributes used in the query, so assume we only have one attribute then we know from above that the possible conditions are:

1. Single simple condition,
2. Single set condition,
3. 2 simple conditions of greater respective less comparator,
4. A $\notin$ set condition and less/greater comparator single condition,
5. A $\neq$ simple condition and less/greater comparator single condition,
6. A $\notin$ set condition and 2 simple conditions of greater respective less comparator.
7. A $\neq$ simple condition and 2 simple conditions of greater respective less comparator.

These are known to be in a canonical form from the earlier lemmas with the exception of the last two. These basically are bounded intervals with a finite set of exception points, and from lemma 4.5 we know that no exception point may be removed. Furthermore would both the set and its complement by infinite in $\mathbb{Q}$ so one could not describe it with a set condition, nor with two less, greater comparators (because the exceptions points). This leaves the option to describe it with a set condition and a less/greater comparator but this is also impossible since the intersection of the complement of the greater/less comparator in the original with less/greater comparator in the new would generate an infinite set which of course could not be removed by a finite exclusion set.

Lets assume that every query with n attributes has a unique minimal canonical form. If we look at a query with $n+1$ attributes, and assume that it does not have a unique minimal canonical form. Then there must exists at least two different minimal

canonical forms (canonical here is just sorted). Now we can split each of those two forms into single attribute queries, and each of those do have a unique minimal canonical form. Furthermore it is evident that the intersection of those single attribute queries would form an equivalent query to the original queries (intersection of two queries can be done by adding the conditions of the two queries). From this it also become evident that the original queries was equal since the two single attribute queries (from each of the queries) are equal (they are in unique minimal canonical form) and each of the original queries are equal to the canonical form of the intersection of their single attribute queries. □

Chapter 5

Minimal canonical form of $\mathcal{L}_{pos}$

When we are using variables (other than the free one) in the query, we immediately run into several problems. First among them is the question if there exists a smaller query which uses the variables differently (other predicates or less of the same predicate). While its quite easy to see that two equivalent queries must use the same predicates (emptying a database instance where the queries was satisfiable of a predicates range which is only in one of the queries), it becomes a lot harder to show that it must be the same number of variables for each predicate.

But before that we define groups of join condition that is easy to work with and shows the impact of the collective join conditions. These are what we are going to call join paths:

Definition 5.1 *(Join path)*
Let Ψ be a *tuple relation formula* and θ be a arithmetic comparison operator then $\langle \Psi, \theta \rangle$ is a join trail between X and Y if any of the following is true:

- Ψ consists only of a join condition of the form $X\theta Y$ and $\theta : <, \leq, =, \geq, >$.
- Ψ consists only of a join condition of the form $Y\theta' X$ where θ' is the reflection of θ and $\theta' : <, \leq, =, \geq, >$.
- Ψ can be separated into Ψ_1 and Ψ_2, where $\langle \Psi_1, \theta_1 \rangle$, $\langle \Psi_2, \theta_2 \rangle$ are join trails from X to Z and Z to Y. Furthermore θ_1 and θ_2 both belongs to either $<, \leq, =$ or $>, \geq, =$. Then $\langle \Psi, \theta \rangle$ is a join trail between X and Y where θ is $<, >$ if either θ_1 or θ_2 is $<$ or $>$, $=$ if both θ_1 and θ_2 are $=$ and $\leq, \geq$ otherwise.

Let $\langle \Psi, \theta \rangle$ be a join trail between X and Y. If $\langle \Psi, \theta \rangle$ contains no cycle then it is called a **join path**.

Unfortunately in $\mathcal{L}_{pos}$ we can have queries like:

$$
\begin{aligned}
&\{x | Person(x) \wedge \\
&\quad (\exists y_1)(\exists y_2)(\exists y_3)(\exists y_4)(\\
&\qquad Knows(y_1) \wedge Knows(y_2) \wedge Person(y_3) \wedge \\
&\qquad Person(y_4) \wedge y_1.knowerID = x.ID \wedge y_2.knowerID = x.ID = x.ID \wedge \\
&\qquad y_1.knowsID = y_3.ID \wedge y_2.knowsID >= y_4.ID \wedge \\
&\qquad y_3.height \neq y_4.height \wedge y_3.weight \neq y_4.weight
\end{aligned}
$$

Which means to get all persons who knows two persons of different height and weight, but since both height and weight are comparable one can sort the persons on either height and weight and then get these two queries:

$$\{x|Person(x)\wedge$$
$$(\exists y_1)(\exists y_2)(\exists y_3)(\exists y_4)($$
$$Knows(y_1)\wedge Knows(y_2)\wedge Person(y_3)\wedge$$
$$Person(y_4)\wedge y_1.knowerID = x.ID\wedge y_2.knowerID = x.ID = x.ID\wedge$$
$$y_1.knowsID = y_3.ID\wedge y_2.knowsID = y_4.ID\wedge$$
$$y_3.height < y_4.height\wedge y_3.weight \neq y_4.weight$$

$$\{x|Person(x)\wedge$$
$$(\exists y_1)(\exists y_2)(\exists y_3)(\exists y_4)($$
$$Knows(y_1)\wedge Knows(y_2)\wedge Person(y_3)\wedge$$
$$Person(y_4)\wedge y_1.knowerID = x.ID\wedge y_2.knowerID = x.ID = x.ID\wedge$$
$$y_1.knowsID = y_3.ID\wedge y_2.knowsID = y_4.ID\wedge$$
$$y_3.height \neq y_4.height\wedge y_3.weight < y_4.weight$$

Both this queries are of the same length using the earlier measure and I am presuming it to be the minimal length since I have not found any smaller query.

Now before we go on with the lemmas, there is important to note that every query in $\mathcal{L}_{pos}$ can be written as a single component query. This is because a positive component consists of only existance quantifiers.

To mitigate this we are going to restrict us to a stricter language and for that we need to define join trees.

Definition 5.2 *(Join tree)*
Begin with the free variable as a root node. A variable is a leaf to a node if it has a join trail of = to the variable representing the node and that trail can not be continued to the parent of the node. If this graph has no cycles then its called a **join forest** and if every node has a path (in the graph) to every other node then its called a **join tree**.

Definition 5.3 *($\mathcal{L}_{pos'}$)*
Let $\mathcal{L}_{pos'}$ be the subset language of $\mathcal{L}_{pos}$ where the only join conditions are $=$, the join conditions form a join forest, every variable is over a distinct predicate and there are no simple or set conditions on any join attributes.

It is quite evident that two equivalent queries in $\mathcal{L}_{pos'}$ has the same enumeration (just enumerate on the lexical order of the predicate names) because they both must have their variables over the same predicates.

Lemma 5.1 *(Minimal canonical form over $\mathcal{L}_{pos'}$)*
Let $l \in \mathcal{L}_{pos'}$ be a query with only one component and that component lacks redundancies, then l has a minimal canonical form.

Proof 5.1 *(Minimal canonical form over $\mathcal{L}_{pos'}$)*
Take $l_1 \in \mathcal{L}_{pos'}$ that is equivalent to l. To realize that they are over the same predicates we take a satisfiable assignment to l and turn it into a database instance. This instance will have one schema tuple in every predicate that is in l, and if l_1 have a predicate that l does not then it would be unsatisfiable in that instance or if it does not have a predicate that is in l we can simply remove that tuple from the database instance making l unsatisfiable. Both of those contradict that l and l_1 would be equivalent so

they must run over the same predicates and this ensures that we can given them both the same enumeration (using the lexical order of the predicate names).

It also evident from the above that we can take a satisfiable assignment to l (order by the enumeration) and it would also be a satisfiable assignment to l_1. Now lets look at the join forests of these two queries and if they are not equal then we can find two variables and attributes which has a join path between them in only one of the queries, say in l. Since a variable attribute in a join condition can not have a simple or set condition, we now that for each value (in $\mathcal{U}$) there exists a satisfiable database instance where the variable attributes has those values. From this we can see that they variable attributes can not have simple or set conditions in l_1, and we also know that there was no join path between them so we can basically take a satisfiable database instance for l and change one of variable attributes values (and propagate it over its join paths). Then it would be satisfiable by l_1 but not by l. This gives that they have the same join forest so we only need to define a deterministic approach to transfer the join forest into a minimal set of join conditions. This is done by enumerating the nodes (with the same enumeration as the variables) and then taking the node with least number and add join conditions to all it was connected to and thereafter removing that node from the join forest (and continue doing this until there are no nodes left).

What now is left are the simple and set conditions in l and l_1 and if we would treat each variable as a $\mathcal{L}_{simple}$ query we know that they have a minimal canonical form, so lets assume that both l and l_1 have their simple and set conditions in minimal canonical form from lemma 4.6. And furthermore we also know that the simple and set conditions are not connected to the join attribute so we can treat them separately for each variable. So if l has a variable with an attribute that has different set or simple condition that those in l_1 then we can look at its allowed values and since we assumed that they both had minimal simple and set conditions we can realize that the set of allowed values must not be equal and then there exists a value which l_1 has but not l (or reversed). That breaks the equivalence so they must have the same simple or set conditions. □

Chapter 6

Implementation

While the implementation is probably easiest understood with giving the rewrite rules and an strategy of application, there are some things that need to be stated. The current implementation is not complete, i.e. there exists equivalent queries which are not reduced to the same form. Some of them are because the merging of variables is done quite late and after it we do not recheck with negative component propagation/cancellation. Others are because that $\neq$ may sometimes be replaced by $<$ on the ground that the join condition was over the same variable type and attribute. There are certainly others by these are the two most significant I've personally found.

Furthermore it has not been proved that the rules (and the implementation) is sound, but I've until now not found any counter examples of it.

6.1 Rewrite rules:

Unrestricted

Commute components: Changing place between two components.

Commute conditions: Changing place between two conditions in a component.

Renumber variable: Renumber the variables in a component.

Reverse join: Rewrite a join $(z_i.attr_1 \theta z_j.attr_2)$ to $(z_j.attr_2 \theta' z_i.attr_1)$ where θ' is the reversed operator for θ.

Positive component merge: Writing two $((\exists \bar{z_1})\Psi_1) \wedge ((\exists \bar{z_2})\Psi_2)$ into $((\exists \bar{z_1})(\exists \bar{z_2}'))(\Psi_1 \wedge \Psi_2')$, where $\bar{z_i}$ is vector of variables, Ψ_i is a conjunctive group of atomic conditions over these variables and the notative $'$ means that we renumber the variables in $\bar{z_2}$ and Ψ_2 so that the variable numbers do not collide with $\bar{z_1}$.

Restricted components rewrites through constraints

Negative component merge: If we have two $\neg((\exists \bar{z})(\Psi \wedge (z_i.attr\theta_1 c_1) \wedge \ldots \wedge (z_i.attr\theta_k c_k))) \wedge \neg((\exists \bar{z})(\Psi \wedge (z_i.attr\theta_1' d_1) \wedge \ldots \wedge (z_i.attr\theta_l d_l)))$ then we can either

$k = 1, l = 1$, $d_1 = c_1$ and there exists an operator that expresses the $\theta : \theta_1 \vee \theta'_1$ relation (e.g. $\geq : < \vee =$). Note in this case d_1 and c_1 may be variable attribute references. Then it can be expressed as $\neg((\exists \overline{z})(\Psi \wedge (z_i.attr\theta c_1)))$.

Otherwise if $\{x|x\theta_1 c_1 \wedge \ldots \wedge x\theta_k c_k\} \cup \{x|x\theta'_1 d_1 \wedge \ldots \wedge x\theta'_l d_l\}$ can be described by a conjunction of regular conditions then we can express it as $\neg((\exists \overline{z})(\Psi \wedge (z_i.attr\theta''_1 e_1) \wedge \ldots \wedge (z_i.attr\theta''_m e_m)))$.

Negative component propagation: If we have $\neg((\exists \overline{z_1})(\Psi_1 \wedge (z_i.attr\theta c))) \wedge ((\exists \overline{z_1})(\exists \overline{z_2})(\Psi'_1 \wedge \Psi_2))$, where Ψ_1 is decidable true when Ψ'_1 is true, then we can rewrite it as $\neg((\exists \overline{z_1})(\Psi_1 \wedge (z_i.attr\theta c))) \wedge ((\exists \overline{z_1})(\exists \overline{z_2})(\Psi_1 \wedge (z_i.attr\theta' c) \wedge \Psi_2))$ where θ' is the negated θ, note that c can here be a set, variable attribute or a constant.

Negative component cancellation: If we have $\neg(((\exists \overline{z_1})(\exists \overline{z_2}))(\Psi_1 \wedge \Psi_2)) \wedge ((\exists \overline{z_1})(\exists \overline{z_3})(\Psi'_1 \wedge \Psi_3))$, where Ψ_1 is decidable true when Ψ'_1 is true, then we can rewrite it as $\neg((\exists \overline{z_2})\Psi_2) \wedge ((\exists \overline{z_1})(\exists \overline{z_3})(\Psi'_1 \wedge \Psi_3))$.

Positive component split: If we have $((\exists \overline{z_1})(\exists \overline{z_2}))(\Psi_1 \wedge \Psi_2)$ where Ψ_i are only over variables from $\overline{z_i}$ (for $i = 1, 2$) then we can rewrite it as $((\exists \overline{z_1})\Psi_1) \wedge ((\exists \overline{z_2})\Psi_2)$.

Restricted condition rewrites through constraints

Join condition merge: If we have $((\exists \overline{z})(\Psi \wedge (z_i.attr_1\theta_1 z_j.attr_2) \wedge (z_i.attr_1\theta_2 z_j.attr_2)))$ then we can rewrite it as $((\exists \overline{z})(\Psi \wedge (z_i.attr1\theta z_j.attr2)))$ where θ is the operator corresponding to the conjunction of θ_1 and θ_2, of course if they are contradictory then the component is not satisfiable.

Simple and/or set condition merge: If we have $((\exists \overline{z})(\Psi \wedge (z_i.attr\theta_1 c_1) \wedge (z_i.attr\theta_2 c_2)))$ then we can rewrite it as $((\exists \overline{z})(\Psi \wedge \Psi_2)$ where Ψ_2 is the result from canonicalizing $z_i.attr\theta_1 c_1 \wedge z_i.attr\theta_2 c_2$.

Simple equality propagation: If we have in a component $z_i.attr_1 = c$ and $z_j.attr_2\theta z_i.attr_1$ or $z_i.attr_1\theta' z_j.attr_2$ then we replace that join condition with $z_j.attr_2\theta c$ (note θ' is the reversed θ).

Join equality propagation: If we have in a component $z_i.attr_1 = z_j.attr_2$, $z_i.attr_1\theta_1 c_1$ and $z_j.attr_2\theta_2 c_2$ then we can also add $z_i.attr_1\theta_2 c_2$ and $z_j.attr_2\theta_1 c_1$.

Join condition propagation: If there in a component exists $z_i.attr_1\theta z_j.attr_2$ and $z_i.attr_1$ has a minimum/maximum value c, where θ is $<, \geq$ respectively $>, \leq$ then we can add $z_j.attr_2\theta' c$ where θ' is the reverse operator of θ.

Join condition redundancy: If there in a component exists $z_i.attr_1\theta_1 z_j.attr_2$ and a join path from $z_i.attr_1$ to $z_j.attr_2$ with length greater than 1 and operator θ_2 where θ_2 is the same as or a stricter variant than θ_1 then we can remove $z_i.attr_1\theta_1 z_j.attr_2$ from the component.

Join paths propagation: If there exists a join path in a component between $z_i.attr_1$ and $z_j.attr_2$ with operator θ then we can introduce $z_i.attr_1\theta z_j.attr_2$ in the component (if it does not exists already).

Restricted rewrites through equivalence checking

Component removal: If $l \wedge \Phi \equiv l$ then remove component Φ.

Variable removal: If $l \wedge \pm((\exists \bar{z}, z_1)(\Psi \wedge \Psi_1)) \equiv l \wedge \pm((\exists \bar{z})\Psi)$ where Ψ_1 had all conditions that was over z_1, then remove z_1.

Variable merge: If $l \wedge \pm((\exists \bar{z}, z_1, z_2)\Psi) \equiv l \wedge \pm((\exists \bar{z}, z_1)\{z_1/z_2\}\Psi)$ then replace z_2 with z_1.

Join removal: If $l \wedge \pm((\exists \bar{z})(\Psi \wedge (z_i.attr_1 \theta z_j.attr_2))) \equiv l \wedge \pm((\exists \bar{z})\Psi)$, then remove the join condition $z_i.attr_1 \theta z_j.attr_2$)

Simple/set removal: If $l \wedge \pm((\exists \bar{z})(\Psi \wedge (z_i.attr_1 \theta c))) \equiv l \wedge \pm((\exists \bar{z})\Psi)$, then remove the simple/set condition $z_i.attr_1 \theta c$)

6.2 Applying the rewrite rules

Unfortunately, applying the rewrite rules non-deterministically will probably not generate the results one are after. So one has to divide the procedure into a couple of steps.

1. Fill component: The first thing to do is for each component in the query merge all simple and set conditions to their canonical form. Then we use join paths propagation until no new join conditions are introduced and after that merge the join condition. There after use join condition propagation (or join equality propagation) for each join condition, and lastly we merge (again) the simple and set conditions to their canonical form.

2. Component merge: Merge every positive component and perform negative component merge until there are no negative components which can be merged together.

3. Component reduction: Perform first negative component propagation and when one can not do that any more perform negative component cancellation.

4. Reduce component: First we merge all simple and set conditions to their canonical form, then we use simple equality propagation (until we can not do it anymore). After that we perform join condition redundancy and join condition merger.

5. Check component: Here we try component removal, then variable removal/merge for each variable in the component. After that each join condition is tested for join removal and each simple/set condition is tested for removal.

6. Sort query: First we enumerate and sort each component and then we order the components.

6.3 STEP

`STEP` is a logical system that has implemented the Q language, and it does also contain an full working implementation of the described system above. As a side note it may be said that this implementation can reduce a query to something that is non-equivalent (in `STEP`), but this has been because (in every noted case) that `STEP` did not enforce a total order on $\mathcal{U}$.

6.3.1 Examples & runs

The following queries are using the three predicates *Moviei*, *IsDirector* and *Entertainer*, and the attributes that are named after the predicate it belongs to followed by ID is consider a primary key. Note though that foreign restrictions are not considered in this query.

$$
\begin{aligned}
&\{m|Movie(m)\wedge\\
&\quad(\exists y_1)(\exists y_2)(\\
&\qquad IsDirector(y_1)\wedge Entertainer(y_2)\wedge y_2.lastName=\text{'Lucas'}\wedge\\
&\qquad m.movieID=y_1.movieID\wedge y_1.entertainerID=y_2.entertainerID)\wedge\\
&\quad m.year>1963\wedge\\
&\quad\neg(\exists y_1)(\exists y_2)(\\
&\qquad IsDirector(y_1)\wedge Entertainer(y_2)\wedge y_2.lastName=\text{'Lucas'}\wedge\\
&\qquad m.title=\text{'Star Wars'}\wedge m.movieID=y_1.movieID\wedge\\
&\qquad y_1.entertainerID=y_2.entertainerID)\wedge
\end{aligned}
$$

This query basically states that one should get all movies from 1963 and onwards directed only by Lucas, which is not Star Wars.

$$
\begin{aligned}
&\{x|Movie(x)\wedge\\
&\quad x.year>1963\wedge\\
&\quad x.title\neq\text{'Star Wars'}\wedge\\
&\quad(\exists y_1)(\exists y_2)(\\
&\qquad IsDirector(y_1)\wedge Entertainer(y_2)\wedge y_1.entertainerID=y_2.entertainerID)\wedge\\
&\qquad x.movieID=y_1.movieID\wedge y_2.lastName=\text{'Lucas'})\wedge
\end{aligned}
$$

The next query is a listing of all director which have not worked with Lucas or Allen on a movie.

$$
\begin{aligned}
&\{p|Entertainer(p)\wedge\\
&\quad\neg(\exists y_1)(\exists y_2)(\exists y_3)(\exists y_4)(\\
&\qquad Entertainer(y_1)\wedge IsDirector(y_2)\wedge IsDirector(y_3)\wedge\\
&\qquad Movie(y_4)\wedge p.entertainerId=y_2.entertainerID\wedge\\
&\qquad y_1.entertainerID=y_3.entertainerID\wedge y_2.movieID=y_4.movieID\wedge\\
&\qquad y_3.movieID=y_4.movieID\wedge y_1.lastName=\text{'Lucas'})\wedge\\
&\quad\neg(\exists y_1)(\exists y_2)(\exists y_3)(\exists y_4)(\\
&\qquad Entertainer(y_1)\wedge IsDirector(y_2)\wedge IsDirector(y_3)\wedge\\
&\qquad Movie(y_4)\wedge p.entertainerId=y_2.entertainerID\wedge\\
&\qquad y_1.entertainerID=y_3.entertainerID\wedge y_2.movieID=y_4.movieID\wedge\\
&\qquad y_3.movieID=y_4.movieID\wedge y_1.lastName=\text{'Allen'})
\end{aligned}
$$

$$
\begin{aligned}
&\{x|Entertainer(x)\wedge\\
&\quad\neg(\exists y_1)(\exists y_2)(\exists y_3)(\exists y_4)(\\
&\qquad Movie(y_1)\wedge IsDirector(y_2)\wedge IsDirector(y_3)\wedge\\
&\qquad Entertainer(y_4)\wedge y_2.movieID=y_1.movieID\wedge\\
&\qquad y_3.movieID=y_1.movieID\wedge y_4.entertainerID=y_2.entertainerID\wedge\\
&\qquad x.entertainerID=y_3.entertainerID\wedge y_4.lastName\in\{\text{'Lucas','Allen'}\})
\end{aligned}
$$

And in here is the last query and its canonicalization, the query is basically asking for all movies that have another movie made by the same director at an same year but after 1950.

$$
\begin{aligned}
&\{m|Movie(m)\wedge\\
&\quad(\exists y_1)(\exists y_2)(\exists y_3)(\exists y_4)(
\end{aligned}
$$

$$
\begin{aligned}
&Entertainer(y_1) \wedge IsDirector(y_2) \wedge IsDirector(y_3) \wedge \\
&Movie(y_4) \wedge m.movieID = y_2.movieID \wedge m.movieID \neq y_4.movieID \wedge \\
&m.year <= y_4.year \wedge m.year >= y_4.year \wedge \\
&y_1.entertainerID = y_2.entertainerID \wedge \\
&y_1.entertainerID = y_3.entertainerID \wedge y_3.movieID = y_4.movieID \wedge \\
&y_4.year > 1950)
\end{aligned}
$$

$$
\begin{aligned}
&\{x | Movie(x) \wedge \\
&\quad x.year > 1950 \wedge \\
&\quad (\exists y_1)(\exists y_2)(\exists y_3)(\exists y_4)(\\
&\qquad Movie(y_1) \wedge IsDirector(y_2) \wedge IsDirector(y_3) \wedge \\
&\qquad Entertainer(y_4) \wedge y_2.movieID = y_1.movieID \wedge \\
&\qquad y_3.entertainerID = y_2.entertainerID \wedge \\
&\qquad y_4.entertainerID = y_2.entertainerID \wedge \\
&\qquad x.year = y_4.year \wedge x.movieID = y_3.movieID \wedge x.movieID \neq y_4.movieID
\end{aligned}
$$

The runtimes of the examples above was 1.7, 3.2 and 3.5 seconds to run, and while these times where quite good there are examples of queries taking above 100 seconds to run through. Though in most of the testcases I've run seems to be below 10 seconds on a UltraSparc II 440 MHz and 256 MB memory.

Chapter 7

Conclusion

While the full goal of this thesis was not reached, there was some interesting result regarding the definition of minimality and canonicality used here. Among them was the result that there are queries which knowingly had multiple minimal canonical forms e.g. queries in $\mathcal{L}_{pos}$ which using the current minimality definition can not have canonical form, because if z_i and z_j is of the same type then $(z_i.attr_1 \neq z_j.attr_1) \wedge (z_i.attr_2 \neq z_j.attr_2)$ is equivalent to $(z_i.attr_1 < z_j.attr_1) \wedge (z_i.attr_2 \neq z_j.attr_2)$, $(z_i.attr_1 \neq z_j.attr_1) \wedge (z_i.attr_2 < z_j.attr_2)$, $(z_i.attr_1 > z_j.attr_1) \wedge (z_i.attr_2 \neq z_j.attr_2)$ and $(z_i.attr_1 \neq z_j.attr_1) \wedge (z_i.attr_2 > z_j.attr_2)$. This might be solved by preferring $\neq$ over $<, >$ join conditions, but then one also have to check its effect on other queries and the algorithms.

Although in the more restricted languages we know that the canonical minimal form is unique, and it is probably also true for somewhat less restricted variants of $\mathcal{L}_{pos'}$, for instance it should be possible for the extension of $\mathcal{L}_{pos'}$ where we allow multiple variables of the same predicate. One of the problems that comes up then is whether there exists equivalent queries which have different number of variables on a certain predicate, and neither of the queries has any redundancies. Furthermore it should also be possible to extend it for more types of joins.

Bibliography

[1] ABITEBOUL, S., VIANNU, R., AND HULL, V. *Foundations of Database Systems*, 3rd ed. Addison Wesley, 1995.

[2] BROWDER, A. *Mathematical Analysis. An Introduction*, 3rd ed. Springer, 2001, pp. 1–20.

[3] ELMASRI, R., AND NAVATHE, S. *Foundations of Database Systems*, 3rd ed. Addison Wesley, 2000, pp. 195–206, 299–308.

[4] MINOCK, M. Knowledge representation using schema tuple queries. *KRDB-03* (Sept. 2003).

[5] MINOCK, M. A phrasal generator for describing relational database queries. *EACL* (Apr. 2003).