Incremental Evaluation of Relational Expressions

Nicolas Barbier

promotor :
Prof. dr. Jan VAN DEN BUSSCHE

Universiteit Maastricht

Eindverhandeling voorgedragen tot het bekomen van de graad
Licentiaat in de informatica afstudeerrichting databases

universiteit hasselt

Since then we have witnessed the proliferation of baroque, ill-defined and, therefore, unstable software systems. Instead of working with a formal tool, which their task requires, many programmers now live in a limbo of folklore, in a vague and slippery world, in which they are never quite sure what the system will do to their programs. Under such regretful circumstances the whole notion of a correct program—let alone a program that has been proved correct—becomes void. What the proliferation of such systems has done to the morale of the computing community is more than I can describe.

Edsger W. Dijkstra

Abstract

This thesis is about the field of relational databases. It investigates techniques that allow to incrementally maintain the results of relational queries when the database content changes. The main use-case is keeping up-to-date *materialized views*, which are queries whose results are stored in the database system. The considered query language is the relational algebra based on set semantics, including the generalized projection operator to express aggregation. Modeling the changes of the database content happens by, for each relation, specifying a set of tuples to delete and a set of tuples to insert. The update that should be applied to the value of a view to keep it up-to-date with the new database content, is modeled in different ways, depending on which technique is under investigation. The models used are *count tables*, *deltas* and *change tables*. Also investigated is the notion of *self-maintainability* of (sets of) views, which expresses whether (sets of) views can be maintained without access to the relations they are based on.

Acknowledgements

I would like to thank the following people for helping me during the writing of this text:

- My promotor Jan Van den Bussche and supervisor Dirk Leinders for wanting to guide me through the process and proofreading my drafts.
- My brother Michael for, despite not being trained in computer science, lending me an ear whenever I felt like explaining to someone what I was doing.
- Tom Monnier for proofreading the (almost-) final version.
- Jo Gielis for confirming how much better it looks to use a 10pt sized font.
- Johan Huysmans for taking care of getting my text submitted.

I hope that your efforts are not wasted this year, and that I will finally graduate `:-)`.

Nicolas

Summary

This is a summary of the content of this thesis in Dutch.

Deze thesis gaat over relationele databases. Ze handelt over het incrementeel onderhouden van de resultaten van relationele algebra uitdrukkingen, als de inhoud van de database verandert. Dit is vooral belangrijk in de context van het up-to-date houden van zogenaamde *materialized views* ("gematerialiseerde views", relationele uitdrukkingen waarvan het resultaat in de database wordt opgeslagen).

In het **eerste hoofdstuk** wordt de gebruikte relationele algebra beschreven, en worden een aantal begrippen ingevoerd die van belang zijn in de rest van de tekst.

Een van de ingevoerde begrippen is de *delta*: dit is een koppel van verzamelingen, genaamd de positieve en de negatieve *change-set* ("veranderingen-verzameling"). Een delta modelleert het verschil tussen twee relatie-instanties van dezelfde ariteit. Een delta is een delta *tussen* twee relatie-instanties als de tweede relatie-instantie het resultaat is van de tupels uit de negatieve change-set te verwijderen uit de eerste relatie-instantie, en dan de tupels uit de positieve change-set toe te voegen.

Voor deltas wordt een belangrijke *normaalvorm* eigenschap ingevoerd: minimaliteit. Een delta is minimaal voor een relatie-instantie, als alle tupels in de positieve change-set nog geen elementen van de relatie-instantie zijn, en alle tupels in de negatieve change-set wel degelijk elementen van de relatie-instantie zijn. Er wordt aangetoond dat er slechts één minimale delta bestaat tussen twee relatie-instanties van dezelfde ariteit, en we dus kunnen spreken van *de* minimale delta tussen twee zulke relatie-instanties.

Geïnverteerde deltas worden ingevoerd, en er wordt aangetoond dat de inverse van een minimale delta een eigenschap bezit die intuïtief noodzakelijk is: het toepassen van een minimale delta op een relatie-instantie, en op het resultaat de inverse delta toepassen, geeft terug de originele relatie-instantie

Ook de term *transactie* wordt in dit hoofdstuk gedefinieerd. Een transactie modelleert het verschil tussen twee database-instanties over hetzelfde schema, en wordt beschouwd als een functie van relatie-namen naar deltas. Een *minimale transactie* is een transactie waarvan alle deltas minimaal zijn.

Hoofdstuk twee introduceert *afgeleide data*. Meer specifiek worden twee specifieke vormen aangehaald: *indices* en *materialized views*. Indices worden niet behandeld in de tekst.

In het geval van een materialized view treedt het volgende probleem op: wanneer de inhoud van de database verandert, kan het zijn dat de opgeslagen waarde niet meer consistent is met de definitie van het view. In dit geval moet

het view up-to-date gebracht worden. Aangezien er meestal van kan worden uitgegaan dat de veranderingen aan de database-inhoud klein zijn ten opzichte van de database-inhoud zelf (de zogenaamde *heuristic of inertia*), is het in vele gevallen efficiënter om niet de hele uitdrukking die het view definieert te herevalueren, maar om de verschillen met de vorige toestand te gebruiken om het verschil tussen de oude toestand van het materialized view en de nieuwe (gewenste) toestand te berekenen. Dit op een efficiënte manier doen wordt ook wel het *view maintenance problem* genoemd.

Een aantal verschillende dimensies worden vermeld, volgens welke men verschillende technieken voor view maintenance kan categoriseren.

In **hoofdstuk drie** wordt het zogenaamde *count algoritme* beschreven. Dit is een algoritme voor view maintenance dat enkel gedefinieerd is voor SPJ-uitdrukkingen. Dit zijn relationele algebra uitdrukkingen, die enkel de operatoren selectie, projectie en Cartesisch product gebruiken, en waarvoor op eenvoudige wijze een normaalvorm gespecifieerd kan worden.

De belangrijkste eigenschap van dit algoritme is dat in het *extent* (de representatie van de waarde van het view) een extra attribuut wordt bijgehouden voor elk tupel van het resultaat. Dit attribuut houdt bij hoeveel *veroorzakende* tupels er in de basisrelaties aanwezig zijn, die het aanwezig zijn van het betreffende tupel in het resultaat verklaren. Dit moet geïnterpreteerd worden in de context van projectie: een tupel dat het resultaat is van een projectie, kan veroorzaakt worden door meerdere tupels in de operand van de projectie. Enkel indien alle veroorzakende tupels uit de operand verwijderd worden, moet ook het tupel uit het resultaat verwijderd worden.

Het count algoritme krijgt een database-instantie en een minimale transactie ervoor als input. Het berekent een zogenaamde *count tabel*, die voor elk te beïnvloeden tupel van het resultaat bevat hoeveel veroorzakende tupels er verwijderd respectievelijk toegevoegd zijn. Indien het resulterende aantal oorzaken nul wordt, wordt het tupel verwijderd. Als een tupel een positief aantal oorzaken wordt toebedeeld, en het maakte nog geen deel uit van het resultaat, dan wordt het toegevoegd.

De details van het count algoritme worden uitgewerkt, en er wordt op een intuïtieve manier bewezen dat het doet waarvoor het is ontworpen; namelijk dat als de berekende count tabel wordt toegepast op het extent dat hoort bij de gegeven database-instantie, dat het resultaat dan het extent is dat hoort bij de nieuwe database-instantie. De nieuwe database-instantie is hier te interpreteren als: de gegeven database-instantie waarop de gegeven transactie is toegepast.

Er worden twee mogelijke uitbreidingen kort aangehaald. Ten eerste is het redelijk eenvoudig om de unie operator toe te laten in de te onderhouden uitdrukkingen. De tweede uitbreidingsmogelijkheid is het toelaten van één aggregatie-operator als buitenste operator van de uitdrukkingen. Deze uitbreidingsmogelijkheden worden niet in detail besproken in de tekst.

Het **vierde hoofdstuk** introduceert een zeer algemeen algoritme voor het onderhouden van views: *algebraïsche delta propagatie*. Deze techniek gebruikt de algebraïsche structuur van de uitdrukking, en *propageert* deltas van de bladeren naar de wortel van de uitdrukkingsboom. De deltas voor de bladeren zijn bekend: dit zijn de deltas die horen bij de transactie. De delta voor de wortel kan worden gebruikt om de waarde van het view aan te passen.

Er wordt een paar van recursieve functies gedefinieerd, dat, gegeven de te onderhouden uitdrukking, een paar van uitdrukkingen berekent. Indien dit paar geëvalueerd wordt, is het resultaat een delta tussen de oude en de nieuwe waarde van het view. Deze functies worden gedefinieerd voor alle relationele operaties behalve aggregatie.

Om de deltas voor elke knoop van de uitdrukkingsboom te berekenen, is het in het algemeen niet genoeg om de deltas horende bij de kinderknopen te kennen; er is ook toegang nodig tot de basisrelaties. Merk op dat hier een keuze gemaakt kan worden: refereren, in de uitdrukkingen die het resultaat zijn van de recursieve functies, de referenties naar de basisrelaties naar de oude (voor de transactie) of de nieuwe (na de transactie) waarde hiervan? Er wordt aangetoond dat, indien men één van de twee mogelijkheden kiest, men de functies op een eenvoudige manier kan aanpassen zodat men de andere mogelijkheid krijgt. Hiervoor worden twee bewijzen geleverd. Het eerste is waarschijnlijk intuïtief duidelijker, het tweede daarintegen is eleganter.

Er wordt bewezen dat de gegeven recursieve functies inderdaad resulteren in uitdrukkingen die een correcte delta berekenen. Hiervoor wordt een omvangrijk bewijs gebruikt dat in appendix A teruggevonden kan worden.

Een notationeel probleem in verband met het algoritme wordt beschreven: gebruikt men de normale algebraïsche notatie voor de uitdrukkingen, dan kunnen de gegenereerde uitdrukkingen exponentieel langer zijn dan de oorspronkelijke uitdrukking. Dit probleem kan worden opgelost door gebruik te maken van een notationeel systeem dat gebruik maakt van afkortingen voor resultaten, die meerdere keren hergebruikt kunnen worden. In zulk een systeem hebben de gegenereerde uitdrukkingen een lengte die lineair is in de lengte van de oorspronkelijke uitdrukking.

Er wordt een complexiteitsanalyse uitgevoerd door middel van een eenvoudige kosten-functie. De uitkomst hiervan is dat, als men de projectie-operator buiten beschouwing laat, de incrementele uitdrukkingen efficiënter uitgevoerd kunnen worden dan de originele uitdrukking, als de veranderingen maar klein genoeg zijn ten opzichte van de grootte van de basisrelaties.

Op het einde van het hoofdstuk bevindt zich nog een deel waarin aggregatie wordt toegevoegd aan de ondersteunde operaties.

In **hoofdstuk vijf** wordt een techniek ingevoerd die *change tabel propagatie* wordt genoemd. De oorspronkelijke, in de literatuur beschreven techniek heeft als doel om zowel aggregatie als *outer joins* te optimaliseren ten opzichte van de delta propagatie techniek. Dit wordt gedaan door een alternatief model te gebruiken om veranderingen voor te stellen. Vanaf het moment dat een aggregatie- of outer join operator wordt tegengekomen tijdens het propagatieproces, wordt in de plaats van een delta een change tabel geproduceerd en verder gepropageerd. Dit werkt enkel als de te onderhouden uitdrukking aan een heel aantal beperkingen voldoet, en is daarom geen volledig algemene techniek.

De hier beschreven versie van de techniek beperkt zich tot het creëren van change tabellen bij aggregatie-operators (outer join operaties worden dus niet rechtstreeks ondersteund). Er wordt echter getracht de techniek op een meer formele manier te benaderen dan in de literatuur.

Vergelijkbaar met de situatie bij het count algoritme wordt er een extent gedefinieerd, dat een extra attribuut bevat dat het aantal oorzaken voor de aanwezigheid van een tupel bevat. Ook de change tabellen bevatten zulk een attribuut, dat wordt gebruikt om het corresponderende attribuut van het extent aan te passen.

Een change tabel bestaat uit een relatie-instantie en een aantal bijkomende eigenschappen. Er wordt een verzameling van *group-by* attributen gedefinieerd, die bepalen welke attributen van een change tabel gebruikt moeten worden om corresponderende tupels in de change tabel en het extent te vinden. Voor de andere attributen worden functies gedefinieerd, die uitdrukken hoe deze attributen in de change tabel gecombineerd moeten worden met de corresponderende attributen in het extent.

Er wordt aangegeven hoe een change tabel gecreëerd kan worden uit een delta en hoe een change tabel te propageren door relationele operatoren.

Hoofdstuk zes is een kort hoofdstuk dat handelt over een eigenschap genaamd *self-maintainability* ("zelf-onderhoudbaarheid"). Deze eigenschap is van belang in het geval de systemen die de basisrelaties en het view bevatten niet transactioneel gekoppeld zijn. Hierdoor kan men, als men de basisrelaties benadert, er niet zeker van zijn dat men de waarde observeert die hoort bij de transactie die men aan het verwerken is. Het kan dus interessant zijn om te onderzoeken welke (verzamelingen van) views onderhouden kunnen worden zonder de waarden van de basisrelaties te gebruiken.

Een view wordt self-maintainable genoemd, als het onderhouden kan worden met enkel kennis van de transactie en de oude waarde van het view. Analoog wordt een verzameling views self-maintainable genoemd, als alle elementen ervan onderhouden kunnen worden met enkel kennis van de transactie en de oude waarden van de views. Observeer dat het van fundamenteel belang is, dat volgens deze definitie voor het onderhoud van een van de elementen van de verzameling, de oude waarden van de andere elementen ook gebruikt mogen worden.

Er wordt aangetoond dat niet alle views self-maintainable zijn, en dat men elke verzameling views zo kan uitbreiden, dat ze self-maintainable wordt.

In **hoofdstuk zeven** worden kort enkele problemen aangehaald die zouden kunnen optreden bij het implementeren van materialized views voor het vrije databasesysteem PostgreSQL.

Hoofdstuk acht tenslotte, bespreekt een aantal gerelateerde onderwerpen. Deze onderwerpen worden niet in detail uitgewerkt in deze thesis.

- Het concept om views *uitgesteld* te onderhouden. Dit beduidt het niet onmiddelijk op de views toepassen van veranderingen aan de basisrelaties.

- Het optimaliseren van het evalueren van relationele uitdrukkingen in de aanwezigheid van materialized views. Aangezien performantie de reden is van het bestaan van materialized views, is het voor de hand liggend dat het interessant zou zijn om een databasesysteem deze optimalisatie automatisch uit te laten voeren.

- Het vaststellen van welke materialized views er best gecreëerd kunnen worden, om een zo hoog mogelijke performantie te garanderen. Dit is het logische vervolg van het vorige punt: ook de keuze welke views te materialiseren zou geautomatiseerd kunnen worden.

Contents

1 Introductory Material **1**
1.1 Database Schemas and Instances 1
1.2 Relational Expressions 1
1.3 Deltas 4
1.4 Transactions 5

2 Incremental View Maintenance **6**
2.1 Derived Data 6
2.2 Classification of Incremental View Maintenance Techniques . . . 7
2.3 All-at-Once vs. Leaf-by-Leaf Propagation 8

3 The Count Algorithm **9**
3.1 Definition 9
3.2 Correctness 11
3.3 Extensions 11
3.3.1 Adding Union 11
3.3.2 Adding Aggregation 12

4 Algebraic Delta Propagation **13**
4.1 Definition 13
4.2 Correctness 15
4.3 Complexity Analysis 16
4.3.1 Query Size 16
4.3.2 Time Complexity Under a Simplistic Cost Model 16
4.4 Adding Aggregation 19

5 Change Table Propagation **21**
5.1 Definition 22
5.2 Aggregation Function Requirements 23
5.3 Creation 24
5.4 Application 24
5.5 Propagation 25
5.6 Correctness 25

6 Self-Maintenance **27**
6.1 Definition 27

7 Case Study: Investigating an Implementation for PostgreSQL **29**

8 Related Topics **30**
8.1 Deferred View Maintenance 30
8.2 Optimizing Queries in the Presence of Derived Data 30
8.3 Choosing Derived Data Structures for Optimal Performance . . . 31

A Proofs **34**

Chapter 1

Introductory Material

1.1 Database Schemas and Instances

Definition 1.1 The countable set of all *primitive values* in a database is denoted by $\mathbb{U}$.

Definition 1.2 A *database schema* is a function from a set of relation names to the natural numbers. These natural numbers are called *arities*.

Database schemas are denoted with bold, uppercase letters starting from $\mathbf{S}$.

Definition 1.3 A *relation instance* of arity n is a finite set of n-tuples over $\mathbb{U}$. The arity of a relation instance i is denoted $\operatorname{ar}(i)$.

Note that the unnamed perspective it used; natural numbers starting from 1 are used to identify the attributes of a tuple. In logical formulas, attribute numbers use a bold typeface to distinguish them from numerical constants.

Definition 1.4 Two relation instances are *compatible* iff they have the same arity.

Definition 1.5 A *database state* or *database instance* of a database schema $\mathbf{S}$ is a function from the relation names in the schema to relation instances. The relation instances should have the arities that are dictated by $\mathbf{S}$. The empty database instance, which is the unique database instance of the empty schema, is written $D_\emptyset$.

Database states are usually denoted using uppercase letters starting from D.

1.2 Relational Expressions

Definition 1.6 A *relational expression* over a database schema $\mathbf{S}$ is an expression constructed using the operators that follow. A relational expression Q can be *evaluated* in the context of a database instance D of $\mathbf{S}$, written $D(Q)$. $D(Q)$ is a relation instance of a certain arity, written $\operatorname{ar}(Q)$. Let Q, Q_1 and Q_2 be relational expressions, D a database instance of $\mathbf{S}$, R a relation name occurring in $\mathbf{S}$, and θ a predicate that ranges over tuples:

- A relation instance i. This expression evaluates to the relation instance itself.

  $$D(i) = i$$

- R. This signifies a relation name in $\mathbf{S}$. The evaluation of R in the context of D, $D(R)$, is the relation instance $D(R)$ itself.

- $\sigma_\theta(Q)$. This signifies the selection of tuples from Q that satisfy condition θ. The arity of the result is the same as that of Q.

  $$D(\sigma_\theta(Q)) = \{q : q \in D(Q) \wedge \theta(q)\}$$

- $\pi_{A,f_1,\dots,f_m}(Q)$. This signifies the generalized projection. Inspiration for this operator is taken from Gupta et al. [GHQ95]. A is a set of attribute numbers, and $f_1, \dots, f_m$ is a list of functions from relation instances to primitive values. The f_i are called *aggregation functions*. To evaluate a generalized projection, first the projection of its argument on A is computed. Then, the result is augmented with, for each f_i, an extra attribute that is the result of applying f_i to the set of tuples that were projected to the tuple in question. The arity of the result is the sum of the number of attribute numbers in A and m.

  $$(q_1, \dots, q_n)|_A = (q_{i_1}, \dots, q_{i_m}), \quad \text{where } \{i_1, \dots, i_m\} = A \text{ and } 1 \leq i_1 < \dots < i_m \leq n$$

  $$D(\pi_{A,f_1,\dots,f_m}(Q)) = \{t|_A \cdot (f_1(g(t, A)), \dots, f_m(g(t, A))) : t \in D(Q)\}, \quad \text{where } g(t, A) = \{u : u \in D(Q) \wedge u|_A = t|_A\}$$

  Examples of aggregation functions are:

  - `sum`(n), the summation operator over the attribute with attribute number n.
  - `count`$(*)$, the operator that returns the number of tuples in its argument.
  - `min`(n), the operator that returns the minimal value of all values for the attribute with attribute number n.
  - `max`(n), the operator that returns the maximal value of all values for the attribute with attribute number n.
  - `avg`(n), the operator that returns the average value of all values for the attribute with attribute number n.

  When no aggregation functions are present in a generalized projection operator, it is just called *projection*. This is important because some algorithms in this text only work on expressions that do not include aggregation. In case aggregation is not allowed, the definition of the operator simplifies to:

  $$D(\pi_A(Q)) = \{t|_A : t \in D(Q)\}$$

- $Q_1 \cup Q_2$, $Q_1 \cap Q_2$, and $Q_1 - Q_2$. These signify the union, intersection, and difference, respectively, of Q_1 and Q_2. The arities of Q_1 and Q_2 should be the same, and the arity of the result equals this arity.

$$\begin{aligned} D(Q_1 \cup Q_2) &= \{q : q \in D(Q_1) \vee q \in D(Q_2)\} \\ D(Q_1 \cap Q_2) &= \{q : q \in D(Q_1) \wedge q \in D(Q_2)\} \\ D(Q_1 - Q_2) &= \{q : q \in D(Q_1) \wedge q \notin D(Q_2)\} \end{aligned}$$

- $Q_1 \times Q_2$. This signifies the Cartesian product of Q_1 and Q_2. The arity of the result is the sum of the arities of Q_1 and Q_2.

$$a \cdot b = (a_1, \ldots, a_n, b_1, \ldots, b_m), \qquad \text{where } (a_1, \ldots, a_n) = a \text{ and } (b_1, \ldots, b_m) = b$$

$$D(Q_1 \times Q_2) = \{q_1 \cdot q_2 : q_1 \in D(Q_1) \wedge q_2 \in D(Q_2)\}$$

The following are a few derived operators:

- $Q_1 \bowtie_\theta Q_2$. This signifies the theta-join of Q_1 and Q_2 on condition θ. The arity of the result is the sum of the arities of Q_1 and Q_2.

$$D(Q_1 \underset{\theta}{\bowtie} Q_2) = D(\sigma_\theta(Q_1 \times Q_2))$$

- $Q_1 \ltimes_\theta Q_2$ and $Q_1 \overline{\ltimes}_\theta Q_2$. These signify the semijoin and anti-semijoin of Q_1 and Q_2 on condition θ. The arity of the result is that of Q_1.

$$\begin{aligned} D(Q_1 \underset{\theta}{\ltimes} Q_2) &= D(\pi_{\{1,\ldots,\mathrm{ar}(Q_1)\}}(Q_1 \underset{\theta}{\bowtie} Q_2)) \\ D(Q_1 \underset{\theta}{\overline{\ltimes}} Q_2) &= D(Q_1 - (Q_1 \underset{\theta}{\ltimes} Q_2)) \end{aligned}$$

Definition 1.7 The predicate $A^=$, with $A \subseteq \{1, \ldots, n\}$ a set of attribute numbers, for relation instances of arity $2n$, denotes:

$$\bigwedge_{i \in A} \mathbf{i} = \mathbf{j}, \text{ where } j = n + i$$

Relational expressions are usually denoted using uppercase letters starting from Q. The algebra defined by these operators is called the *relational algebra* (*with aggregation*).

Note that any relational expression that does not include relation names can be considered an expression over the empty schema $\emptyset$. Its evaluation in the context of $D_\emptyset$ is therefore permitted. Usually, the evaluation syntax is not used in this case, and the expression is implicitly considered to be evaluated. For example, to indicate the result of the union of two relation instances, $\{1\} \cup \{2\}$ might be written instead of $D_\emptyset(\{1\} \cup \{2\})$.

Definition 1.8 Two relational expressions Q_1 and Q_2 over a common database schema $\mathbf{S}$ are *equivalent*, iff for all database instances D of $\mathbf{S}$, $D(Q_1) = D(Q_2)$. Equivalence of Q_1 and Q_2 is written $Q_1 \equiv Q_2$.

1.3 Deltas

A *delta* models the difference between two compatible relation instances. When *applied to* the first relation instance, the result is the second relation instance. A delta is represented as a positive change-set (usually prefixed by a $\triangle$-symbol in writing) and a negative change-set (usually prefixed by a $\bigtriangledown$-symbol).

Definition 1.9 A delta is a pair of relation instances $(\triangle i, \bigtriangledown i)$. The result of the application of this delta to a relation instance i is $(i - \bigtriangledown i) \cup \triangle i$. The inverse of this delta, written $(\triangle i, \bigtriangledown i)^{-1}$, is $(\bigtriangledown i, \triangle i)$. A delta *between* relation instances i_1 and i_2 is any delta that, when applied to i_1, yields i_2.

According to this definition, first the negative change-set is deleted from the relation, and only then the positive change-set is inserted. Whether to do it that way, or the other way around, can indeed yield different results. This leads to many kinds of problems:

Example 1.10 Applying a delta to a relation instance, and then applying the inverse on the result, does not necessarily yield the original relation instance.

Proof For the proof, a contradiction is constructed. Consider the following:

$$\begin{aligned} &i = \emptyset \\ &(\triangle i, \bigtriangledown i) = (\emptyset, \{1\}) \end{aligned}$$

This is a contradiction because applying the delta, and then applying its inverse, doesn't yield the original relation instance:

$$(((i - \bigtriangledown i) \cup \triangle i) - \triangle i) \cup \bigtriangledown i = (((\emptyset - \{1\}) \cup \emptyset) - \emptyset) \cup \{1\} = \{1\} \neq \emptyset \qquad \square$$

In order to make the notion of a delta more useful, a concept called *minimality* is introduced, which tries to canonicalize the notion:

Definition 1.11 A delta $(\triangle i, \bigtriangledown i)$ is called *minimal* with respect to application to a relation instance i iff the following properties hold:

$$\begin{aligned} &\bigtriangledown i - i = \emptyset \\ &\triangle i \cap i = \emptyset \end{aligned}$$

Thus, minimality means that no tuples are deleted that are not in the relation instance, and that no tuples are added that are already present. Note that a delta can be minimal for some relation instances, although it is not minimal for others.

The notion of minimality was introduced by Qian et al., although Blakeley et al. captured the property without explicitly naming it [QW91, BLT86].

Remark 1.12 Let $(\triangle i, \bigtriangledown i)$ be a minimal delta between any two relation instances. $\triangle i$ and $\bigtriangledown i$ are always disjoint:

$$\triangle i \cap \bigtriangledown i = \emptyset$$

Proof Let i be the relation instance to which $(\triangle i, \bigtriangledown i)$ is applied.

$$\begin{aligned} & \forall x : (x \notin \bigtriangledown i \vee x \in i) \wedge (x \notin \triangle i \vee x \notin i) \\ \Longrightarrow\ & \forall x : x \notin \triangle i \vee x \notin \bigtriangledown i \\ \Longleftrightarrow\ & \triangle\, i \cap \bigtriangledown i = \emptyset, \end{aligned}$$

which concludes the proof. □

Remark 1.13

1. Any two compatible relation instances have exactly one minimal delta between them. Therefore, one can unambiguously refer to *the* minimal delta between two compatible relation instances.

2. The minimal delta between two relation instances i_1 and i_2 is $(i_2 - i_1, i_1 - i_2)$.

Proof A proof for this remark can be found in appendix A.

Remark 1.14

1. Applying a delta to a relation instance for which it is minimal, yields a relation instance for which the inverse delta is minimal.

2. Also, applying the inverse delta on the result of the previous, yields the original relation instance.

Proof Let i_1 be the original relation instance, $(\triangle i, \bigtriangledown i)$ the minimal delta for it, and $i_2 = (i_1 - \bigtriangledown i) \cup \triangle i$ the result of applying $(\triangle i, \bigtriangledown i)$ to i_1. Applying the result of remark 1.13 to the delta between i_2 and i_1, yields:

$$\begin{aligned} \triangle i &= i_1 - i_2 \\ \bigtriangledown i &= i_2 - i_1 \end{aligned}$$

...which coincides with $(\triangle i, \bigtriangledown i)^{-1}$, also according to remark 1.13. Therefore, $(\bigtriangledown i, \triangle i)$ must be the minimal delta between i_2 and i_1. □

1.4 Transactions

Definition 1.15 A *transaction* between two database states D_{old} and D_{new} of a database schema $\mathbf{S}$ is a function that maps every relation name R in $\mathbf{S}$ to a delta between $D_{\text{old}}(R)$ and $D_{\text{new}}(R)$. The *inverse* of a transaction t is the transaction corresponding to the function that maps to the inverse deltas, and is written t^{-1}.

Transactions are usually denoted using lowercase letters starting from t.

Definition 1.16 A minimal transaction is a transaction that maps to minimal deltas only.

Definition 1.17 The *delta collection* deltas(t) of a transaction t for a database schema $\mathbf{S}$ is a database instance that maps, for any $R \mapsto (\triangle i, \bigtriangledown i)$ in t, two relation names $\triangle R$ and $\bigtriangledown R$ to $\triangle i$ and $\bigtriangledown i$, respectively. It is assumed that all relation names $\triangle R$ and $\bigtriangledown R$ do not conflict with relation names in $\mathbf{S}$.

Chapter 2

Incremental View Maintenance

2.1 Derived Data

Derived data is data that is defined in terms of other data. Major examples in the field of relational database systems include *materialized views* and *indices*. Derived data is used to increase the performance of database systems, by providing the data in a form that is potentially more suitable for the execution of the operations at hand.

In relational databases, a *view* is a relational expression that has been assigned a name, which can be used in queries as if it were a normal relation. A view is called a *derived relation*, while normal relations are called *base relations*. Adding the concept of views to a database system is straightforward: whenever a view reference occurs in a query during execution, the view reference is replaced by its definition, and the resulting expression is the one that is really executed.

A *materialized view* is a view whose extent (a representation of the *value* of the view; i.e., a representation of the result of evaluating the expression that defines the view) is stored in the database itself. It can be seen as a cache for the result of the view definition. When an expression refers to a materialized view, it can be evaluated by directly using this cache, thereby avoiding the computations that would otherwise be necessary to compute the result of the view's defining expression. Of course, the base relations that a materialized view is based on can change. Like a cache, the materialized view should then be *invalidated*. Bringing a materialized view up to date with the current database state is called *view maintenance*. The problem of doing this in a performant way is called the *view maintenance problem*.

Indices are data structures that accelerate the process of determining which tuples in a certain relation satisfy certain predicates. Being a form of derived data, indices too need to be kept consistent with the data they are derived from.

Incremental Maintenance of Derived Data The most straightforward way to do view maintenance is simply recomputing the whole materialized view whenever it needs to be brought up to date. However, in most cases one can do better: most data in the base relations and view will not have changed. This is called the *heuristic of inertia* by Gupta et al. [GM95]. One could try to optimize the situation by only computing the necessary changes, and applying them to the materialized view. Several algorithms to do this exist, and some of them are investigated in this thesis.

Concerning indices, most database systems provide specialized incremental algorithms to keep them in sync with the relations they are defined on. Those are not investigated in this text.

2.2 Classification of Incremental View Maintenance Techniques

Gupta et al. distinguish the following dimensions by which incremental view maintenance algorithms can be classified [GM95]:

Information Dimension This refers to the information needed to calculate the modifications to apply.

- The relations that the view is based on. It may be possible to have access to the state of these as before or after the modifications took place. Those two situations are differentiated in this text by means of the *pre-update* and *post-update* terminology, introduced in section 4.1. It may also be possible that these states are both unavailable, but the difference between one of them and the available state becomes available afterwards. This may be the case when the transaction updating the view is decoupled from the transaction that makes the changes to the base relations.
- The materialized view itself. If access to the materialized view might be needed, but access to the base relations is not, the view maintenance technique is known as a *self-maintenance* technique. Chapter 6 investigates this topic.
- The integrity constraints that the base relations adhere to. This text pays no attention to that information. One may want to consult Vista's PhD thesis for an investigation of this topic [Vis97].

Modification Dimension This dimension refers to what modifications to the base tables are possible, and the way they are modeled. Examples could be that modifications are expressed as tuples to insert or delete (e.g., the *deltas* introduced in section 1.3). They could be expressed as `SQL`'s `INSERT` and `DELETE` statements. Updates to tuples could be modeled as insertions followed by deletions, as incremental changes (e.g., "add 5 to attribute a of tuple x"), or otherwise. Schema changes may or may not be possible. It may be possible to change the view definition, and still reuse the old data in the materialized view to speed up the generation of the new data.

In this thesis, modifications on the base relations are modeled as deltas. The modifications to apply to the materialized views are modeled in a number of ways. Changes to the view definition or database schema are not considered. For more information on the latter, see Gupta et al. [GMR95].

Language Dimension The language used to express the view definition. Examples are the relational algebra, the Chronicle query languages (see Jagadish et al.), Datalog, etc… [JMS95] It also includes properties such as whether set- or bag-semantics are used, and whether aggregation is allowed or not.

The language used in this thesis is the relational algebra with set semantics. Aggregation is also considered. For some maintenance algorithms, the allowed operators are restricted, or expressions having specific properties are not allowed.

Instance Dimension This refers to what database instances and modification instances the algorithm applies to. An example is that some algorithms restrict the modifications to insertions of tuples, and disallow updates and deletions. This might for example be useful in a data warehousing context.

None of the algorithms in this thesis restrict the database instances on which they can operate, nor do they disallow any modifications that can be expressed in the change model.

2.3 All-at-Once vs. Leaf-by-Leaf Propagation

Many incremental maintenance algorithms propagate changes from the leaves of the expression tree (i.e., references to relations), up to the top level (i.e., the result). For most of these techniques, one can differentiate between two approaches. Although most of the existing literature does not state this explicitly, one of these approaches is usually adhered to:

All leaves at once The changes for all leaves are processed in one run. The algorithm must be able to combine changes in both operands of a binary operator. This usually makes the algorithms somewhat more complex.

Leaf-by-leaf The changes for the leaves are processed leaf by leaf. The algorithm only needs to be able to propagate changes from one operand of a binary operator at a time. This may simplify the algorithm, but it may also decrease its performance. For example, one cannot always easily incorporate optimizations that exploit integrity constraints, as the intermediate database states might violate them.

Note that this works even when the same relation reference is used multiple times in the expression. In that case one can just consider them as referring to different relations that initially have the same content, and also end up having the same new content after all leaves have been processed.

In this text, both approaches are used. The first one is the preferred approach, and the second one is used if the complexity of the technique would otherwise be too high.

Chapter 3

The Count Algorithm

The *count algorithm* is a rather simplistic and restricted approach to incremental view maintenance. It is an *all-at-once* (see section 2.3) algorithm that is defined for SPJ-expressions only. The result of the algorithm is a *count table*, which represents the changes to be applied to the *extent* (the representation of the value of the view). To compute this count table, access is needed to:

- The deltas applied to the base relations.
- The values of the base relations before the transaction took place.

A count table can be used in a straightforward way to update the old value of the view to the new value.

The name of this algorithm is derived from the fact that a number is stored for every row in the extent. This number represents the number of reasons for the row to exist. Whenever this count reaches zero, the row is to be deleted from the view.

3.1 Definition

Blakeley et al. describe an algorithm for the incremental maintenance of views defined by SPJ-expressions [BLT86]. SPJ-expressions are relational algebra expressions limited to the operations *select*, *project*, and *Cartesian product* (the 'J' in SPJ actually stands for *join*, but here the term is defined with Cartesian product instead).

Lemma 3.1 Every SPJ-expression is equivalent to an expression of the form:

$$\pi_A(\sigma_\theta(R_1 \times \cdots \times R_n))$$

This form is called the *canonical form* of an SPJ-expression.

Proof The proof can be found in appendix A.

The input to the algorithm is a database instance D, a minimal transaction t for D, and a view definition V. The output indicates what changes are to be applied to the view. It is a set of tuples, in which each tuple is tagged with a non-zero integer number. Intuitively, the number represents how many *causes* for the tuple to be present are added to or removed from the view. This is to be interpreted in the context of projection, where a resulting tuple can be *caused by* one or more tuples in the original relation.

Definition 3.2 Fix a database instance D. In the context of the count algorithm, the *extent* $\mathcal{E}_{\text{count}}(D, V)$ of a materialized view defined by an SPJ-expression with canonical form $V = \pi_A(Q)$ is defined as:

$$\mathcal{E}_{\text{count}}(D, V) = D\left(\pi_{A,\texttt{count(*)}}(Q)\right)$$

Clearly, an extent is a relation instance in which every tuple has a non-zero natural number as its last component. It is a special representation for a value of the view. One can infer from an extent the current value of the view, as stated in the following remark:

Remark 3.3 Let D be a database instance. Given an extent e of a view defined by V for D, $D(V)$ is given by:

$$\pi_{\{1,\ldots,\text{ar}(V)\}}(e)$$

Proof Let V be equivalent to $\pi_A(Q)$. It is clear that $\text{ar}(V) = |A|$. Thus follows:

$$\pi_{\{1,\ldots,\text{ar}(V)\}}(e) \equiv \pi_{\{1,\ldots,|A|\}}(e) \equiv \pi_{\{1,\ldots,|A|\}}(\pi_{A,\texttt{count(*)}}(Q)) \equiv \pi_A(Q) \equiv V\square$$

Definition 3.4 A *count table* for an extent with arity n is a relation instance of arity n, in which every tuple has a non-zero integer number as its last component.

A count table represents the changes that have to be applied to the extent of a view, whenever it needs to be updated.

Definition 3.5 The *count refresh* operator $\mathcal{R}_{\text{count}}$ operates on an extent e and a count table c for it as follows. Let a be the arity of e.

$$\mathcal{R}_{\text{count}}(c, e) = D_{\emptyset}\left(\sigma_{\mathbf{a}\neq 0}\left(\pi_{\{1,\ldots,a-1\},\texttt{sum}(a)}((e \times \{1\}) \cup (c \times \{2\}))\right)\right)$$

The count refresh operator is the operator that applies the changes, represented by a count table, to the extent of a view.

Definition 3.6 The *count algorithm* takes as input:

- an SPJ-expression $V = \pi_A(\sigma_\theta(R_1 \times \cdots \times R_n))$ (i.e., it is in canonical form),
- a database state D,
- a minimal transaction t for D.

The output is a count table for the extent of V, which is defined by:

$$\triangle J = \bigcup_{i=1}^{n} \prod_{j=1}^{n} \begin{cases} \triangle R_j, & \text{if } i = j \\ (R_j - \triangledown R_j) \cup \triangle R_j, & \text{otherwise} \end{cases}$$

$$\triangledown J = \bigcup_{i=1}^{n} \prod_{j=1}^{n} \begin{cases} \triangledown R_j, & \text{if } i = j \\ R_j, & \text{otherwise} \end{cases}$$

$$\sigma_{\mathbf{a}\neq 0}\left(\pi_{A,\texttt{sum}(c)}(\sigma_\theta(\triangle J) \times \{1\} \cup \sigma_\theta(\triangledown J) \times \{-1\})\right),$$
$$\text{where } c = \text{ar}(R_1) + \cdots + \text{ar}(R_n) + 1 \text{ and } a = |A| + 1$$

This expression is to be evaluated in the context of $D \cup \text{deltas}(t)$.

3.2 Correctness

The following theorem states that the count algorithm is *correct*:

Theorem 3.7 Let $V = \pi_A(Q)$ be an SPJ-expression in canonical form, and t the minimal transaction between any two database states D_{old} and D_{new} of a database schema $\mathbf{S}$. The count algorithm yields a count table c for which the following holds:

$$\mathcal{R}_{\text{count}}\left(c, \mathcal{E}_{\text{count}}\left(D_{\text{old}}, V\right)\right) = \mathcal{E}_{\text{count}}(D_{\text{new}}, V)$$

In other words, it yields a count table that, when applied to the extent of V for D_{old}, yields the view's extent for D_{new}.

Proof The proof is expressed in an intuitive way only:

- If a tuple is in the positive change-set for one of the relations, that causes this tuple to be added to the Cartesian product, multiplied by the *new* contents of all other operands of the Cartesian product. This multiplication has to be in the order of the Cartesian product in the view definition. I.e., if $\triangle R_i$ contains a tuple r_i, then $R_1^m \times \cdots \times R_{i-1}^m \times \{r_i\} \times R_{i+1}^m \times \cdots \times R_n^m$ is added to the Cartesian product, where R_k^m stands for $(R_k - \triangledown R_k) \cup \triangle R_k$.
- If a tuple is in the negative change-set for one of the relations, this causes this tuple, multiplied by the *old* contents of all other operands of the Cartesian product, to be deleted from the Cartesian product. This multiplication has to be in the order of the Cartesian product in the view definition. I.e., if $\triangledown R_i$ contains a tuple r_i, then $R_1 \times \cdots \times R_{i-1} \times \{r_i\} \times R_{i+1} \times \cdots \times R_n$ is deleted from the Cartesian product.
- No other tuples are added to or deleted from the Cartesian product.

This is exactly what the expressions defining $\triangle J$ and $\triangledown J$ express.

These tuples are then filtered by the selection condition θ, to make sure that only tuples that are relevant for the result are considered. Next, the tuples that are to be added to the Cartesian product are tagged with 1, and the ones that are to be deleted from it are tagged with -1. The result is then projected onto the attributes of the projection, and for each resulting tuple the sum of the tags is taken. This sum corresponds to the change in the number of causes for the tuple in question, as a tuple tagged with 1 indicates the addition of a cause, and a tuple tagged with -1 indicates the deletion of a cause. $\square$

3.3 Extensions

3.3.1 Adding Union

It is quite straightforward to extend the count algorithm to additionally support the union operator. This is based on the observation that union operators can always be propagated to the top of the expression, so that a canonical representation for such queries can be defined: it is the union of a set of canonical SPJ-expressions. Such an expression can be maintained by computing count tables for all SPJ subexpressions, and combining them by summing the causes.

This extension is not treated in detail in this text.

3.3.2 Adding Aggregation

It is rather straightforward to extend the count algorithm so that it is able to cope with one generalized projection operator as the outermost operator, if all used aggregation functions are distributive.

Chapter 4

Algebraic Delta Propagation

The technique of *algebraic delta propagation* is a very general one: for every node in the expression tree, a minimal delta is computed, based on the computed deltas for the operands. This delta expresses the difference between the *old* and the *new* value of the subexpression starting at the considered node. The process is started at the leaf-nodes: for these the deltas are known, as they correspond to the deltas in the transaction. *Propagation functions* are defined, that result in expressions that compute the delta for a node, given the deltas for the operands. The end result is the delta for the root node, which can be applied to the extent of the view.

In general, algebraic delta propagation can be studied as a *leaf-by-leaf* or as an *all-at-once* algorithm (see section 2.3). This text describes the all-at-once approach, as it is not significantly more complex.

To compute the delta that is to be applied to the view's extent, access is needed to:

- The deltas applied to the base relations.
- The values of the base relations before or after the transaction took place.

Qian et al. are the first to describe the technique, formulating it for the relational algebra under set semantics [QW91]. Their paper contains some incorrect statements, however, that are only corrected years later by Griffin et al. [GLT97]. Another paper by Griffin et al. describes the same technique under bag semantics [GL95].

4.1 Definition

Definition 4.1 Let D_{old} and D_{new} be database states of a database schema **S**. Let t be the minimal transaction between D_{old} and D_{new}. Let $(\triangle Q, \triangledown Q)$ be a pair of relational expressions that may refer to relation names in **S** and deltas(t). Given a relational expression Q, $(\triangle Q, \triangledown Q)$ is an *incremental maintenance expression* for Q if it calculates a minimal delta between $D_{\text{old}}(Q)$ and $D_{\text{new}}(Q)$.

- If the expressions are to be evaluated in the context of deltas$(t) \cup D_{\text{old}}$, then $(\triangle Q, \bigtriangledown Q)$ is a *pre-update* expression.
- If the expressions are to be evaluated in the context of deltas$(t) \cup D_{\text{new}}$, then $(\triangle Q, \bigtriangledown Q)$ is a *post-update* expression.

This definition makes an interesting difference between pre-update and post-update expressions. It may seem that the choice of evaluating the expressions *before* (pre-update) or *after* (post-update) the transaction is executed, is of fundamental importance. The following theorem though, states that the choice does not matter that much:

Theorem 4.2 For every pre-update expression, an equivalent post-update expression exists. Also, for every post-update expression, an equivalent pre-update expression exists.

Proof Consider every base relation R that is referenced in a pre-update expression for a transaction t. Let $t(R) = (\triangle R, \bigtriangledown R)$. Then, R equals $(R - \triangle R) \cup \bigtriangledown R$ in a post-update expression. To construct an equivalent post-update expression, replace all references to a base relation R with $(R - \triangle R) \cup \bigtriangledown R$.

Analogously, consider every base relation R that is referenced in a post-update expression for a transaction t. Let $t(R) = (\triangle R, \bigtriangledown R)$. Then, R equals $(R - \bigtriangledown R) \cup \triangle R$ in a pre-update expression. To construct an equivalent pre-update expression, replace all references to a base relation R with $(R - \bigtriangledown R) \cup \triangle R$. $\square$

The following proof is a more elegant alternative to the previous proof. A similar proof is mentioned by Colby et al. [CGL+96].

Proof (alternative) Let $(\triangle Q, \bigtriangledown Q)$ be a pre-update expression for Q. Let D_{old} and D_{new} be any two database states, and t the transaction between them. When $(\triangle Q, \bigtriangledown Q)$ is evaluated in the context of $D_{\text{old}} \cup$ deltas(t), the result is a minimal delta between $D_{\text{old}}(Q)$ and $D_{\text{new}}(Q)$. Also, when $(\bigtriangledown Q, \triangle Q)$ is evaluated in the context of D_{new} and deltas(t^{-1}), that yields a minimal delta between $D_{\text{new}}(Q)$ and $D_{\text{old}}(Q)$, which is an inverse minimal delta between $D_{\text{old}}(Q)$ and $D_{\text{new}}(Q)$. Thus, one can construct a post-update expression equivalent with Q by inverting t, and inverting the resulting delta.

Constructing a pre-update expression from a post-update expression can be done analogously. $\square$

The algorithm proposed by Griffin et al. propagates deltas from the base relations to the top-level of the view expression [GLT97]. It is a set semantics version of an algorithm described previously [GL95]. It considers all relational operators except for aggregation (i.e., generalized projection is not allowed to have any aggregation parameters). This is accomplished by a pair of mutually recursive functions $\triangle$ and $\bigtriangledown$, that propagate the positive and negative changesets through relational operators. They take a view definition as parameter. The pair of their results is an incremental maintenance expression, in which plain relation names refer to the relation instances as before the transaction takes place. Thus, it is a pre-update expression.

$\triangle$ and $\bigtriangledown$ are defined in table 4.1. The following abbreviations are used:

$$\begin{aligned}\text{sub}(Q) &= Q - \bigtriangledown(Q)\\ \text{mod}(Q) &= (Q \cup \triangle(Q)) - \bigtriangledown(Q)\end{aligned}$$

V	$\triangle(V)$
i	$\emptyset$
R	$\triangle R$
$\sigma_\theta(Q)$	$\sigma_\theta(\triangle(Q))$
$\pi_A(Q)$	$\pi_A(\triangle(Q)) - \pi_A(Q)$
$Q_1 \cup Q_2$	$(\triangle(Q_1) - Q_2) \cup (\triangle(Q_2) - Q_1)$
$Q_1 \cap Q_2$	$(\triangle(Q_1) \cap \text{mod}(Q_2)) \cup (\triangle(Q_2) \cap \text{mod}(Q_1))$
$Q_1 - Q_2$	$(\triangle(Q_1) - \text{mod}(Q_2)) \cup (\bigtriangledown(Q_2) \cap \text{sub}(Q_1))$
$Q_1 \times Q_2$	$(\triangle(Q_1) \times \text{mod}(Q_2)) \cup (\text{mod}(Q_1) \times \triangle(Q_2))$

V	$\bigtriangledown(V)$
i	$\emptyset$
R	$\bigtriangledown R$
$\sigma_\theta(Q)$	$\sigma_\theta(\bigtriangledown(Q))$
$\pi_A(Q)$	$\pi_A(\bigtriangledown(Q)) - \pi_A(\text{mod}(Q))$
$Q_1 \cup Q_2$	$(\bigtriangledown(Q_1) - \text{mod}(Q_2)) \cup (\bigtriangledown(Q_2) - \text{mod}(Q_1))$
$Q_1 \cap Q_2$	$(\bigtriangledown(Q_1) \cap Q_2) \cup (\bigtriangledown(Q_2) \cap Q_1)$
$Q_1 - Q_2$	$(\bigtriangledown(Q_1) - Q_2) \cup (\triangle(Q_2) \cap Q_1)$
$Q_1 \times Q_2$	$(\bigtriangledown(Q_1) \times Q_2) \cup (Q_1 \times \bigtriangledown(Q_2))$

Table 4.1: Mutually recursive delta propagation functions $\triangle$ and $\bigtriangledown$. Taken from Griffin et al. [GLT97].

4.2 Correctness

To make proving the correctness of the described algorithm easier, a lemma is introduced:

Lemma 4.3 Let D_{ev} and D_{new} be two database states, and Q, $\triangle Q$ and $\bigtriangledown Q$ relational expressions. Then:

$$\begin{aligned}&D_{\text{new}}(Q) = D_{\text{ev}}((Q - \bigtriangledown Q) \cup \triangle Q)\\ &D_{\text{ev}}(\bigtriangledown Q - Q) = \emptyset\\ &D_{\text{ev}}(\triangle Q \cap Q) = \emptyset\end{aligned}$$

. . . is equivalent to: $\forall x$:

$$\begin{aligned}x \in D_{\text{ev}}(\triangle Q) &\iff x \notin D_{\text{ev}}(Q) \wedge x \in D_{\text{new}}(Q)\\ x \in D_{\text{ev}}(\bigtriangledown Q) &\iff x \in D_{\text{ev}}(Q) \wedge x \notin D_{\text{new}}(Q)\end{aligned}$$

Proof The proof for this lemma can be found in appendix A.

In the context of incremental evaluation, these equivalences express that a tuple should be in the positive change-set of a minimal delta iff it was not in Q before the transaction took place, but was in Q after it. Analogously, a tuple should be in the negative change-set of a minimal delta iff it was in Q before, and not in Q after.

The following theorem states that the algebraic delta propagation algorithm is *correct*:

Theorem 4.4 Let V be any relational expression without aggregation, and t the minimal transaction between any two database states D_{old} and D_{new} for a common database schema. $(\triangle(V), \bigtriangledown(V))$—with functions $\triangle$ and $\bigtriangledown$ taken from table 4.1—yields a minimal delta between $D_{\text{old}}(V)$ and $D_{\text{new}}(V)$, when evaluated in the context of $D_{\text{old}} \cup \text{deltas}(t)$. In other words, it is a pre-update incremental maintenance expression for V.

Proof This theorem's proof can be found in appendix A.

4.3 Complexity Analysis

Relational algebra queries can be executed in polynomial time with respect to the size of the input database instance. This is clearly also true for the queries generated by the algebraic delta propagation technique, as they are relational algebra queries.

4.3.1 Query Size

What is more interesting is how the generated queries relate to the query to be incrementally evaluated. Because of the fact that some of the expressions in table 4.1 refer more than once to the functions $\triangle$ and $\bigtriangledown$ themselves, the generated queries may be exponentially longer than the input query. This would be unacceptable for all but the shortest input queries. However, this is merely a syntactic problem: the increase in query length is caused by the repetition of subexpressions. If a syntactic system is used that allows to define *temporary results* or *variables*, and allow for these to be re-used multiple times, the problem does not occur. The length of the generated queries expressed in such a system is linear with respect to the length of the input query.

4.3.2 Time Complexity Under a Simplistic Cost Model

As optimizing relational algebra queries is an undecidable problem, this investigation of the time complexity shall not assume that either the input query, or the corresponding incremental queries are executed in some optimal way. Instead, a rather simplistic cost-model is used to suggest that, given that some conditions are fulfilled, executing the incremental queries is more efficient than executing the input query. This technique is also used by Griffin et al. to suggest the plausibility of the delta propagation technique for the relational algebra with bag semantics [GL95].

The cost model is based on simply executing the operators used in the expression, modeling the cost of each operator. The idea behind the proof is to use a specific property of the incremental expressions. This property is:

- All references to the base relations are in the scope of an intersection or difference operator that bounds the cost of evaluation within the order of the size of the delta relations.

This can be implemented by using one of the operands to *drive* the computation, and *looking up* these tuples in the other one, to decide whether a tuple should be in the result or not. The following lemma states that, given some assumptions, a lookup can be done in constant time:

Lemma 4.5 Let the following assumptions be true:

- Looking up a tuple in a relation (i.e., determining whether the tuple is in the relation or not) can be done in constant time.
- Q is a relational expression without aggregation, and without projection.

Then, looking up a tuple in Q can be done in constant time.

Proof The proof is by induction on the structure of Q:

Relation reference and constant relation This is given as an assumption.

Selection If the tuple passes the condition, the result is the result of doing a lookup on the operand. Otherwise, the result is `false`.

Union The result is the disjunction of doing lookups on both operands.

Intersection The result is the conjunction of doing lookups on both operands.

Difference The result is `false` if the result of a lookup on the first operand is `false`. Otherwise, the result is the result of a lookup on the second operand.

Cartesian product The result is the conjunction of doing a lookup on the first operand for a prefix of the tuple, and doing a lookup on the second operand for a postfix of the tuple, chosen so that the arities correspond to the arities of the operands.

As all the operations described can, by induction, be done in constant time, it follows that a lookup can be done in constant time. □

Furthermore, the next lemma states that this lookup operation makes it possible to evaluate an intersection or difference operator in time linear in the number of tuples in the left operand:

Lemma 4.6 Given the same assumptions of lemma 4.5, executing an intersection or difference operator takes time that is linear in the number of tuples in the left operand.

Proof This proof is by construction. For both operators, loop over the elements of the left operand. For every such tuple, do a lookup in the right operand. Whether the tuple considered is part of the result depends on the operator, and can be determined as follows:

Intersection A tuple is in the result, iff it is in the right operand.

Difference A tuple is in the result, iff it is *not* in the right operand.

It is clear that this algorithm computes the correct result. Also, because of lemma 4.5, it takes time linear in the number of tuples in the left operand. □

Let the number of tuples in the base relations and in the delta relations be bounded by b and d, respectively. Cost functions $c_{\texttt{ev}}$ and $c_{\texttt{inc}}$ are defined recursively on the structure of the expression, that represent the cost of evaluating a query, and the cost of evaluating its incremental maintenance expressions, respectively. These functions are defined by table 4.2. They are based on the following ideas:

Q	$c_{\texttt{ev}}(Q, b)$	$c_{\texttt{inc}}(Q, b, d)$
i	1	0
R	b	d
$\sigma_\theta(Q')$	$c_{\texttt{ev}}(Q', b)$	$c_{\texttt{inc}}(Q', b, d)$
$Q_1 \cup Q_2$	$c_{\texttt{ev}}(Q_1, b) + c_{\texttt{ev}}(Q_2, b)$	$c_{\texttt{inc}}(Q_1, b, d) + c_{\texttt{inc}}(Q_2, b, d)$
$Q_1 \cap Q_2$	$c_{\texttt{ev}}(Q_1, b) + c_{\texttt{ev}}(Q_2, b)$	$c_{\texttt{inc}}(Q_1, b, d) + c_{\texttt{inc}}(Q_2, b, d)$
$Q_1 - Q_2$	$c_{\texttt{ev}}(Q_1, b) + c_{\texttt{ev}}(Q_2, b)$	$c_{\texttt{inc}}(Q_1, b, d) + c_{\texttt{inc}}(Q_2, b, d)$
$Q_1 \times Q_2$	$c_{\texttt{ev}}(Q_1, b) c_{\texttt{ev}}(Q_2, b)$	$O((c_{\texttt{ev}}(Q_1, b) + c_{\texttt{ev}}(Q_2, b))$ $(c_{\texttt{inc}}(Q_1, b, d) + c_{\texttt{inc}}(Q_2, b, d)))$

Table 4.2: The cost functions $c_{\texttt{ev}}$ and $c_{\texttt{inc}}$

Relation reference or constant relation Scanning a relation takes time linear in the number of tuples that it contains. The cost is considered to be this number of tuples. In the case of a constant relation, the cost is considered to be a random constant.

Selection Filtering the operand does not change the order of the complexity. The cost is just the cost of the operand.

Union Taking the union of the two operands does not change the order of the complexity. The cost is therefore considered to be the sum of the costs of the operands.

Intersection or difference For the special *bounding* operators described before, an implementation according to the proof of lemma 4.6 is assumed, and thus the cost of executing one of these operators is considered the cost of executing the left operand only. For the others, the cost is considered the sum of the costs of executing the operands.

Cartesian product The cost is considered to be the product of the costs of the operands.

Note: The last item may seem strange at first, but it is induced by the fact that the size of the result of a computation and its cost are considered the same in the cost model. This may not always be correct, and shows the suggestive nature of this complexity analysis.

Theorem 4.7 Given a query Q and a transaction t for a given schema.

$$\lim_{b/d \to \infty} \frac{c_{\mathtt{inc}}(Q, b, d)}{c_{\mathtt{ev}}(Q, b)} = 0$$

In other words, using the delta propagation technique is by any factor more efficient than the approach in which the original query is executed, given the assumptions, and given that one uses a large enough database relative to the size of the deltas.

Proof This theorem's proof can be found in appendix A.

4.4 Adding Aggregation

The previous algorithm does not support aggregation. To remedy that, Quass introduced an extension to it that provides aggregation [Qua96]. What follows is a set semantics version of that extension.

As in the previous section, the functions $\triangle(V)$ and $\bigtriangledown(V)$ are derived for a view defined by V. The result is contained in table 4.3. The main work is done by the following queries that calculate a pair $(\triangle^w(Q, A, f_1, \ldots, f_n)$, $\bigtriangledown^w(Q, A, f_1, \ldots, f_n))$.

$$\begin{aligned}
\triangle^w(Q, A, f_1, \ldots, f_n) &= \pi_{A, f_1, \ldots, f_n} \Big(\mathrm{mod}(Q) \underset{A=}{\ltimes} (\triangle(Q) \cup \bigtriangledown(Q)) \Big) \\
\bigtriangledown^w(Q, A, f_1, \ldots, f_n) &= \pi_{A, f_1, \ldots, f_n} \Big(Q \underset{A=}{\ltimes} (\triangle(Q) \cup \bigtriangledown(Q)) \Big)
\end{aligned}$$

This pair is considered to perform the following changes:

- Delete the tuples that might need to be changed because there is a tuple in the incoming delta that has the same values for the projection attributes.
- Insert newly calculated versions for the above tuples, unless there are no tuples left that would lead to a tuple with those values for the projection attributes. Also, insert extra tuples for the projection attributes that are in the delta for the subexpression, but were not yet in the result.

The pair cannot be considered a minimal delta, as it might delete a tuple, and insert a tuple that is exactly the same, which would violate minimality. An example is the insertion of a tuple with a neutral value for all aggregated attributes (e.g., 0 in case of $\mathtt{sum}(n)$), assuming that at least one tuple already existed with the same values for the projection attributes. To make a minimal delta out of the pair, these tuples have to be eliminated. This is exactly what happens in table 4.3.

Theorem 4.8 Let V be any relational expression with or without aggregation, and t the minimal transaction between any two database states D_{old} and D_{new} for a common database schema. $(\triangle(V), \bigtriangledown(V))$—with functions $\triangle$ and $\bigtriangledown$ taken from table 4.1 and table 4.3—yields a minimal delta between $D_{\text{old}}(V)$ and $D_{\text{new}}(V)$, when evaluated in the context of $D_{\text{old}} \cup \text{deltas}(t)$.

Proof This theorem is *not* proved in this thesis.

V	$\triangle(V)$
$\pi_{A,f_1,\ldots,f_n}(Q)$	$\triangle^w(Q, A, f_1, \ldots, f_n) - \bigtriangledown^w(Q, A, f_1, \ldots, f_n)$

V	$\bigtriangledown(V)$
$\pi_{A,f_1,\ldots,f_n}(Q)$	$\bigtriangledown^w(Q, A, f_1, \ldots, f_n) - \triangle^w(Q, A, f_1, \ldots, f_n)$

Table 4.3: Extension of the functions $\triangle$ and $\bigtriangledown$ that adds support for aggregation.

Chapter 5

Change Table Propagation

Situations exist where algebraic delta propagation for doing aggregation can be improved by dropping the requirement that changes are always represented by a delta. To address this concern, Gupta et al. introduce another way of modeling changes, called *change tables* [GM99]. Such change tables are used to represent changes that are the result of aggregation and outer join operations. In this text, outer join operations are not considered, which also considerably simplifies the algorithm. Furthermore, this text tries to give a more formal reinterpretation of (this part of) the technique.

Change tables cannot represent all kinds of changes, so the method is defined as one that is complementary to the algebraic delta propagation technique. The idea is to use deltas until an aggregation operator is encountered. At this point, the deltas are converted to change tables, which are then propagated further up the expression tree. If a change table cannot be further propagated (because of restrictions that are defined in the next sections), one can use the (generally less performant) method described in section 4.4 instead.

At some point in the expression tree, every delta has to be converted to a change table. Therefore, for the technique to be defined for some expressions, *dummy* projections may have to be inserted, that project onto all attributes (i.e., no-ops). This is because projections can be used to convert a delta to a change table.

While the previously treated techniques are in this text defined using the *all-at-once* approach (see section 2.3), the change table propagation technique is defined using the *leaf-by-leaf* approach. The reason for this is that the (otherwise required) interaction between deltas and change tables would become very complex.

To compute the change table that is to be applied to the extent of the view, access is required to:

- The deltas applied to the base relations.
- The values of the base relations before the transaction took place.

5.1 Definition

Definition 5.1 For a view expression to be valid for the change table propagation technique, some of its expression tree's nodes need to be marked as *conversion points*. Only (generalized) projection operators can be conversion points. Every path from the root to a leaf in the expression tree requires exactly one of these conversion points.

Consider an expression Q. Whether Q's outermost operator is a conversion point is written $\mathcal{C}(Q)$.

At these nodes, deltas are converted to change tables. Note that it may be required to add *dummy* projection operators to the view expression, to make sure that every path can be marked.

The representation of the value of the view is similar to the one used in chapter 3 (i.e., the tuples have an extra attribute that stores a number of causes), although the definition of the number of causes is somewhat more complex.

Definition 5.2 Let D be a database instance and V a query with aggregation. In the context of change table propagation, the *extent* $\mathcal{E}_{\text{change}}(D, V)$ of a materialized view defined by V is a relation instance of arity $|V| + 1$, and is defined as:

$$\mathcal{E}_{\text{change}}(D, V) = D(\tau(V))$$

Function τ is used to transform V, and is defined in table 5.1. It adds the count attribute at the conversion points, and combines, for nodes above (i.e., closer to the root than) the conversion points, the counts of the children. For selection the counts are left alone, for projection the counts for all tuples projecting onto one tuple are summed, and for Cartesian product the counts are multiplied.

The counts are defined such that they become zero when the tuple has to be deleted during maintenance. This follows intuitively from the following observations:

- Only when all tuples that project onto the same group-by attributes are deleted, is the tuple in the result that has the same values for the group-by attributes to be deleted.
- If a tuple in one if the operands of a Cartesian product has a number of causes of zero, all tuples in the result stemming from this tuple are to be deleted.

Definition 5.3 A *change table* $\square Q$ for an expression Q, having arity n, over a database schema $\mathbf{S}$, for a database instance D, consists of:

- A relation instance $\square Q$ of arity $n + 1$ (note the double use of $\square Q$, as both the syntax for a change table and this relation instance). The last component of every tuple in $\square Q$ is an integer number, possibly zero.
- A set of attribute numbers $\square_{\mathcal{G}} Q \subseteq \{1, \ldots, n\}$ (the *group-by* attributes).
- Functions $\square_{\mathcal{U}_i} Q : \mathbb{U} \times \mathbb{U} \to \mathbb{U}$ for $1 \leq i \leq n - |\square_{\mathcal{G}} Q|$ (the *update functions*).

V	$\tau(V)$
$\sigma_\theta(Q)$	$\sigma_\theta(\tau(Q))$
$\pi_{A,f_1,\ldots,f_n}(Q)$	$\begin{cases} \pi_{A,f_1,\ldots,f_n,\mathtt{sum}(\mathrm{ar}(Q)+1)}(\tau(Q)) & \text{if not } \mathcal{C}(V) \\ \pi_{A,f_1,\ldots,f_n,\mathtt{count}(*)}(Q) & \text{if } \mathcal{C}(V) \end{cases}$
$Q_1 \times Q_2$	$\pi_{Q_{12},\mathbf{q_1 q_2}}(\tau(Q_1) \times \tau(Q_2))$ where $Q_{12} = \{1,\ldots,q_2\} - \{q_1,q_2\}$ and $q_1 = \mathrm{ar}(Q_1)+1$ and $q_2 = q_1 + \mathrm{ar}(Q_2)+1$

Table 5.1: The function τ, that transforms the query that defines a view into the query that defines its extent

A change table represents the changes that have to be applied to the extent of a view, whenever it needs to be updated. The group-by attributes are a key for the view, and are used to identify which tuples are to be updated (or inserted, or deleted). The last component of every tuple indicates the change in the number of causes for the tuple identified by the group-by attributes. The update functions specify how the non-group-by attributes of a tuple from the change table should be used to update the corresponding attributes of a tuple from the extent.

5.2 Aggregation Function Requirements

The change table propagation algorithm requires that all used aggregation functions have specific properties and are only used in a specific way. Intuitively, it should be the case that the results are only combined using the aggregation function itself, and yield a result as if the aggregation function was applied to the (multi-)set that is the union of the original values that yielded the intermediate results. Also, it is required that the original values can be *inverted*, so that if a value is present together with its inverted value, they cancel each other out.

Definition 5.4 The function $\mathtt{combine}_f$ operates on two values a_1 and a_2 that are the results of aggregation function f on sets A_1 and A_2 ($A_1 \cap A_2 = \emptyset$), and produces the value that is the result of applying f on $A_1 \cup A_2$.

For example, $\mathtt{combine}_{\mathtt{sum}(n)}(a_1, a_2) = a_1 + a_2$.

Definition 5.5 The function $\mathtt{cancel}_V$ operates on a tuple t that is in the input of a generalized projection operator $V = \pi_{A,f_1,\ldots,f_n}$. It produces a tuple that, when added to a set B ($t \in B$), *cancels* the effect of the already existing tuple t in B. This means that $V(B - \{t\}) = V(B \cup \{\mathtt{cancel}_V(t)\})$.

Note that $\mathtt{combine}_f$ and $\mathtt{cancel}_V$ are not defined for all aggregation functions f or generalized projection operators V. Only when they are defined, is the change table propagation technique defined for these aggregation functions and generalized projection operators.

5.3 Creation

A change table can be created for the result of a generalized projection operator, given a minimal delta for the operand.

Definition 5.6 Let $V = \pi_{A,f_1,\ldots,f_n}(Q)$ be a relational expression for which to derive a change table, and $(\triangle Q, \triangledown Q)$ a minimal delta for Q. Let the f_i be distributable aggregation functions. Let c be $\mathrm{ar}(Q)+1$. The change table $\square V$ for V can be calculated as follows:

$$\square V = \pi_{A,f_1,\ldots,f_n,\mathtt{sum}(c)}((\triangle Q \times \{1\}) \cup (\triangledown Q^{-1} \times \{-1\}))$$
$$\text{where } \triangledown Q^{-1} = \{\mathtt{cancel}_{\pi_{A,f_1,\ldots,f_n}}(t) : t \in \triangledown Q\}$$
$$\square_{\mathcal{G}} V = A$$
$$\square_{\mathcal{U}_i} V = \mathtt{combine}_{f_i}$$

5.4 Application

Definition 5.7 The *change table refresh operator* $\sqcup$ operates on an extent $\mathcal{E}_{\text{change}}(D,Q)$ for a view Q and a change table $\square Q$ for it as follows:

$$\mathcal{E}_{\text{change}}(D,Q) \sqcup \square Q = (\mathcal{E}_{\text{change}}(D,Q) - \mathcal{D}) \cup \mathcal{U} \cup \mathcal{N}$$

. . . with. . .

$$\mathcal{D} = \mathcal{E}_{\text{change}}(D,Q) \underset{\square_{\mathcal{G}} Q^{=}}{\rtimes} \square Q$$
$$\mathcal{U} = \sigma_{\mathbf{c}\neq 0}(\pi_{\square_{\mathcal{G}} Q, \square_{\mathcal{U}_1} Q(\mathcal{I}_{\mathbf{1}},\mathcal{I}'_{\mathbf{1}}),\ldots,\square_{\mathcal{U}_a} Q(\mathcal{I}_{\mathbf{a}},\mathcal{I}'_{\mathbf{a}}),\mathbf{c}+\mathbf{c'}}(\square Q \underset{\square_{\mathcal{G}} Q^{=}}{\bowtie} \mathcal{E}_{\text{change}}(D,Q)))$$
$$\mathcal{N} = \square Q \underset{\square_{\mathcal{G}} Q^{=}}{\overline{\rtimes}} \mathcal{E}_{\text{change}}(D,Q)$$

. . . where:

- $c = \mathrm{ar}(Q)+1$ (the number of the count attribute in the first tuple),
- $c' = 2c$ (the number of the count attribute in the second tuple),
- $a = \mathrm{ar}(Q) - |\square_{\mathcal{G}} Q|$ (the number of aggregate attributes),
- $\mathcal{I}_i = |\square_{\mathcal{G}} Q| + 1$ (the number of the attribute containing the result of aggregation function i, in the first tuple) and
- $\mathcal{I}'_i = c + |\square_{\mathcal{G}} Q| + 1$ (idem, in the second tuple).

Intuitively, the following happens. Let *corresponding tuples* in the extent and the change table be tuples that have pairwise equal values for the group-by attributes (i.e., tuples a and b for which $\square_{\mathcal{G}} Q^{=}(a \cdot b)$ holds):

- All tuples that can possibly be influenced by the change table, being all tuples in the extent that have a corresponding tuple in the change table, are deleted from the extent. The set of these is $\mathcal{D}$.

- For all tuples in the change table that have a corresponding tuple in the extent: They are combined with their corresponding tuple (using the update functions), and the resulting tuples are inserted in the extent (i.e., the tuples in the extent are *updated*, as the old values are deleted by the first step). Causes are combined by summing them. If the combination of two tuples results in a tuple that has zero causes, then it is not inserted (i.e., those tuples are *deleted*, as the values are deleted by the first step). The set of tuples that are thus inserted is $\mathcal{U}$.
- All tuples in the change table that do not have a corresponding tuple in the extent are inserted in the extent. The set of these is $\mathcal{N}$.

5.5 Propagation

The propagation of a change table is subject to some strong restrictions. Table 5.2 lists the conditions and how to propagate a change table in case the conditions are met.

V	$\square V$, $\square_{\mathcal{G}} V$, $\square_{\mathcal{U}_i} V$	conditions
$\sigma_\theta(Q)$	$\sigma_\theta(\square Q)$, $\square_{\mathcal{G}} Q$, $\square_{\mathcal{U}_i} Q$	θ does only refer to attributes in $\square_{\mathcal{G}} Q$.
$\pi_{A,f_1,\ldots,f_n}(Q)$	$\pi_{A,f_1,\ldots,f_n,\texttt{sum}(c)}(\square Q)$, $\{1,\ldots,\lvert A\rvert\}$, $\square_{\mathcal{U}_{b_i}} Q$	$A \subseteq \square_{\mathcal{G}} Q$. Each f_i refers to only one attribute with attribute number a_i ($a_i \notin \square_{\mathcal{G}} Q$, $a_i \neq c$), and $\square_{\mathcal{U}_{b_i}} Q = \texttt{combine}_{f_i}$, with $b_i = \lvert\{1,\ldots,a_i\} - \square_{\mathcal{G}} Q\rvert$ and $c = \mathrm{ar}(Q) + 1$
$Q_1 \times Q_2$	$\pi_{Q_{12},\mathbf{q_1 q_2}}(\square Q_1 \times Q_2)$, $\square_{\mathcal{G}} Q_1 \cup \{\lvert Q_1\rvert + i : i \in \square_{\mathcal{G}} Q_2\}$, $\begin{cases} \square_{\mathcal{U}_i} Q_1 & \text{if } i \leq \mathcal{A}_1 \\ \square_{\mathcal{U}_{i-\mathcal{A}_1}} Q_2 & \text{otherwise} \end{cases}$	No conditions are imposed. Q_{12}, q_1 and q_2 are defined as in table 5.1. $\mathcal{A}_1 = \lvert Q_1\rvert - \lvert\square_{\mathcal{G}} Q_1\rvert$. Propagating through the right operand is analogous.

Table 5.2: How to propagate change tables through relational operators. In the second row, a_i is the attribute number of the b_ith non-group-by attribute of $\square Q$, and c is the attribute number of the count attribute of $\square Q$. In the third row, q_1 and q_2 are the numbers of the count attribute in the first and second tuple, respectively. $\mathcal{A}_1$ is the number of aggregation functions in the first tuple.

5.6 Correctness

Theorem 5.8 Let V be a relational expression with aggregation, for which the change table propagation technique is defined, and t the minimal transaction between any two database states D_{old} and D_{new} for a common database schema. $\mathcal{E}_{\text{change}}(D_{\text{old}}, V) \sqcup \square V$, where $\square V$ is the change table for V corresponding to t, yields $\mathcal{E}_{\text{change}}(D_{\text{new}}, V)$.

Proof This theorem is *not* proved in this thesis.

Chapter 6

Self-Maintenance

A view (or set of views) is *self-maintainable* if it (or they) can be maintained without accessing the base relations. This may be important in case the views are materialized in a system that is decoupled from the system that contains the base relations. In such a setting, it may be true that access to the base relations is not possible in a transactionally correct way. Therefore, it is interesting to investigate when (sets of) views are self-maintainable. That is what this chapter is about.

Another approach to this problem is to change the algorithms so that they can cope with seeing a snapshot of the base relations that is ahead-of-time, and fixing-up the view when the related changes arrive. This text does not investigate that approach.

6.1 Definition

Definition 6.1 Let **S** be a database schema and V a relational expression over it. The view defined by V is called *self-maintainable*, if a function f exists for which the following holds: For any two database states D_{old} and D_{new} of S, and t the minimal transaction between them:

$$D_{\text{new}}(V) = f(t, D_{\text{old}}(V))$$

In other words, a view is self-maintainable if:

- the new value for the view can be determined from the combination of its old value and the transaction.

Remark 6.2 Some views are not self-maintainable.

Proof Consider the view defined by $\pi_{\emptyset}(R)$, with R a unary relation that is the only relation in the schema over which the view is defined. Consider two initial database states D_1 and D_2 that map R to $\{(a)\}$ and $\{(a),(b)\}$, respectively. For both of these database states, the view evaluates to $\{()\}$. Consider a transaction that maps R to a delta $(\emptyset, \{(a)\})$ (which is minimal for both D_1 and D_2). When it is applied to D_1, the new value of the view becomes $\emptyset$. However, when it is applied to D_2, the new value is $\{()\}$ (i.e., the value doesn't change). For a view to be self-maintainable, its new value should be fully determined by the old value and the transaction. As this is not the case for the considered view, it follows that not all views are self-maintainable. □

Remark 6.3 A view of the form R, with R a relation name in $\mathbf{S}$, is self-maintainable.

Proof Let f be defined as:

$$f(t, R) = (R - \triangledown R) \cup \triangle R, \text{ where } t(R) = (\triangle R, \triangledown R)$$

The fact that t is a transaction between D_{old} and D_{new} gives:

$$\begin{aligned} D_{\text{new}}(R) &= (D_{\text{old}}(R) - \triangledown R) \cup \triangle R \\ &= f(t, D_{\text{old}}(R)), \end{aligned}$$

which concludes the proof. □

Definition 6.4 Let $\mathbf{S}$ be a database schema, and D_{old} and D_{new} any two database states of it. Let t be the transaction between D_{old} and D_{new}. A set of views $\{V_1, \ldots, V_n\}$, defined by expressions over $\mathbf{S}$ is called *self-maintainable* if, for $i \in \{1, \ldots, n\}$, functions f_i exist, such that:

$$D_{\text{new}}(V_i) = f_i(t, D_{\text{old}}(V_1), \ldots, D_{\text{old}}(V_n))$$

In other words, a set of views is self-maintainable if the new values of the views can be determined from only the transaction and the old values. Note that a set of views can be self-maintainable, although not all of its elements are self-maintainable when considered separately. It follows that a set may not be self-maintainable, although a superset of it may be. As the following theorem states, a set of views can always be augmented so that it becomes self-maintainable.

Theorem 6.5 For any set of views V, a set $V' \supseteq V$ exists, such that V' is self-maintainable.

Proof The intuition behind the proof is that, if one adds all used base relations to the set of views (remember that base relations are self-maintainable according to remark 6.3), these base relations can be used to recalculate any relational expressions over them.

Let $V' = \{V'_1, \ldots, V'_n\}$ be $V \cup I_V$, with I_V the set of views defined by, for each base relation used in V, that base relation itself.

Let f_i be defined as follows:

$$f_i(t, R_1, \ldots, R_n) = \begin{cases} (R_i - \triangledown R_i) \cup \triangle R_i, & \text{if } R_i \in I_V \\ V_i^*, & \text{otherwise} \end{cases}$$
$$\text{where } t(R_i) = (\triangle R_i, \triangledown R_i)$$

Here, V_i^* stands for V'_i with all occurrences of base relations $V'_j \in I_V$ replaced by $f_j(t, R_1, \ldots, R_n)$. Intuitively, every occurrence of a base relation is replaced by an expression calculating the *new* version of it.

For the f_i with $V'_i \in I_V$, the same logic as in the proof of remark 6.3 applies. Therefore, the following is true for the f_i with $V'_i \notin I_V$:

$$\begin{aligned} D_{\text{new}}(V'_i) &= D_{\text{old}}(V_i^*) \\ &= f_i(t, V'_i, \ldots, V'_n), \end{aligned}$$

which concludes the proof. □

Chapter 7

Case Study: Investigating an Implementation for PostgreSQL

The following problems are potential show-stoppers for the implementation of the techniques described in this text for the free software database system PostgreSQL. It is clear that the techniques should first be adapted to a bag-version of the relational algebra.

- The implementation of the set operators does not have the properties regarding complexity, stated in lemma 4.5. The set operators intersection and difference always take time linear in the sum of the sizes of both operands. This may lead to performance problems.
- The concurrency control system that is used by PostgreSQL (a form of MVCC, multi-version concurrency control, first described by Bernstein et al. [BG83]), does not allow for real serializable execution of transactions, unless one makes use of full-table locks. This may also lead to performance problems.
- There is no direct support for abbreviating temporary results in the syntactic system (which is a variant of SQL). This will lead to the exponential blow-up of the expression size mentioned in section 4.3.1 if the technique is implemented in the straightforward way.

There is support for triggers, which can be used to detect changes to the base relations. There is also experimental support for triggering on commits, which may be used to implement some more advanced techniques for deferred view maintenance (see section 8.1).

Chapter 8

Related Topics

8.1 Deferred View Maintenance

The previous sections describe algorithms for determining the changes that have to be applied to a view, given the changes applied to the base relations for one transaction. This seems to suggest that view maintenance should be performed at the end of every transaction that modified the base relations. However, depending on the workload, that is not always the most performant approach. If, for example, many changes are performed on the base relations in between the executions of queries that use the view, it might be more performant to batch together all those changes, and calculate and apply one (probably bigger) change to the view.

This technique is called *deferred view maintenance.* In Colby et al. and He et al., techniques are explored that have to do with *when* to apply or calculate *which* changes, and how to *store* the values of the deltas to the base relations in the meantime [CGL$^+$96, HXYY05].

8.2 Optimizing Queries in the Presence of Derived Data

In the presence of materialized views, it may be possible to formulate queries that execute faster by directly referring to those materialized views in the query expression. This, however, puts the burden of query optimization on the query writer. The problem of automating the process of rewriting a query on the base relations into an equivalent—but more performant—query that uses materialized views, is usually classified under the general *query optimization problem.* Also, the traditional problem of optimizing queries in the presence of indices falls under this problem.

8.3 Choosing Derived Data Structures for Optimal Performance

Under the previous approaches, a database administrator has to choose which derived data structures to create. Going even further, this choice could be automated. The problems of doing this for indices and materialized views are called the *index selection problem* and the *view selection problem*, respectively. The inputs for the problems are a database schema, a database workload (a set of queries and the frequency of their executions), and optionally some constraints such as a storage limit. An algorithm that solves the problem would then output which indices or materialized views should be instantiated. Another approach is to continuously scan the queries being executed, and adapting the indices to create and views to materialize while the system is running. For this, see Kotidis et al. [KR99].

One area that is heavily investigated, is the sub-problem of choosing materialized views that can be used for the evaluation of queries over data warehouses organized in *star-schemas*. Usually, the problem is simplified to choosing which combinations of the dimensional attributes are to be aggregated over. Those combinations are usually represented as nodes in a lattice that expresses a *can-be-derived-from* relationship. The corresponding problem of optimizing queries in such a schema, given a set materialized views, is a much lighter problem than the general problem described in the previous section.

Bibliography

[BG83] Philip A. Bernstein and Nathan Goodman. Multiversion concurrency control - theory and algorithms. *ACM Trans. Database Syst.*, 8(4):465–483, 1983.

[BLT86] José A. Blakeley, Per-Åke Larson, and Frank Wm. Tompa. Efficiently updating materialized views. In Carlo Zaniolo, editor, *Proceedings of the 1986 ACM SIGMOD International Conference on Management of Data, Washington, D.C., May 28-30, 1986*, pages 61–71, New York, NY, USA, 1986. ACM Press.

[CGL$^+$96] Latha S. Colby, Timothy Griffin, Leonid Libkin, Inderpal S. Mumick, and Howard Trickey. Algorithms for deferred view maintenance. In *SIGMOD '96: Proceedings of the 1996 ACM SIGMOD international conference on Management of data*, pages 469–480, New York, NY, USA, 1996. ACM Press.

[GHQ95] Ashish Gupta, Venkatesh Harinarayan, and Dallan Quass. Generalized projections: A powerful approach to aggregation. In *VLDB'95, Proceedings of 21th International Conference on Very Large Data Bases, September 11-15, 1995, Zurich, Switzerland*, 1995.

[GL95] Timothy Griffin and Leonid Libkin. Incremental maintenance of views with duplicates. In *SIGMOD '95: Proceedings of the 1995 ACM SIGMOD international conference on Management of data*, pages 328–339, New York, NY, USA, 1995. ACM Press.

[GLT97] Timothy Griffin, Leonid Libkin, and Howard Trickey. An improved algorithm for the incremental recomputation of active relational expressions. *Knowledge and Data Engineering*, 9(3):508–511, 1997.

[GM95] Ashish Gupta and Inderpal S. Mumick. Maintenance of materialized views: Problems, techniques and applications. *IEEE Quarterly Bulletin on Data Engineering; Special Issue on Materialized Views and Data Warehousing*, 18(2):3–18, 1995.

[GM99] Himanshu Gupta and Inderpal S. Mumick. Incremental maintenance of aggregate and outerjoin expressions, 1999.

[GMR95] Ashish Gupta, Inderpal S. Mumick, and Kenneth A. Ross. Adapting materialized views after redefinitions. In Michael J. Carey and

Donovan A. Schneider, editors, *Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data, San Jose, California, May 22-25, 1995*, pages 211–222. ACM Press, 1995.

[HXYY05] Hao He, Junyi Xie, Jun Yang, and Hai Yu. Asymmetric batch incremental view maintenance. In *ICDE '05: Proceedings of the 21st International Conference on Data Engineering (ICDE'05)*, pages 106–117, Washington, DC, USA, 2005. IEEE Computer Society.

[JMS95] H. V. Jagadish, Inderpal S. Mumick, and Abraham Silberschatz. View maintenance issues for the chronicle data model. In *Proceedings of the Fourteenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, May 22-25, 1995, San Jose, California*, pages 113–124. ACM Press, 1995.

[KR99] Yannis Kotidis and Nick Roussopoulos. Dynamat: A dynamic view management system for data warehouses. In Alex Delis, Christos Faloutsos, and Shahram Ghandeharizadeh, editors, *SIGMOD 1999, Proceedings ACM SIGMOD International Conference on Management of Data, June 1-3, 1999, Philadelphia, Pennsylvania, USA*, pages 371–382. ACM Press, 1999.

[Qua96] Dallan Quass. Maintenance expressions for views with aggregation. In *VIEWS*, pages 110–118, 1996.

[QW91] Xiaolei Qian and Gio Wiederhold. Incremental recomputation of active relational expressions. *Knowledge and Data Engineering*, 3(3):337–341, 1991.

[Vis97] Dimitra Vista. *Optimizing incremental view maintenance expressions in relational databases.* PhD thesis, University of Toronto, 1997.

Appendix A

Proofs

Proof of Remark 1.13 This proof is by construction. Let i_1 and i_2 be two relation instances. For a minimal delta $(\triangle i, \triangledown i)$ between them, the following statements must hold:

$$\begin{aligned}
&(i_1 - \triangledown i) \cup \triangle i = i_2 \\
&\triangledown i - i_1 = \emptyset \\
&\triangle i \cap i_1 = \emptyset
\end{aligned}$$

These statements are equivalent to, $\forall x$:

$$\begin{aligned}
&(x \in i_1 \land x \notin \triangledown i) \lor x \in \triangle i \iff x \in i_2 \\
&x \notin \triangledown i \lor x \in i_1 && \text{(i)} \\
&x \notin \triangle i \lor x \notin i_1 && \text{(ii)}
\end{aligned}$$

From which follows:

$$\begin{aligned}
&((x \in i_1 \land x \notin \triangledown i) \lor x \in \triangle i \iff x \in i_2) \\
\iff &(((x \in i_1 \land x \notin \triangledown i) \lor x \in \triangle i) \land x \notin i_1 \iff x \in i_2 \land x \notin i_1) \\
\iff &(x \in \triangle i \land x \notin i_1 \iff x \in i_2 \land x \notin i_1) \\
\overset{\text{(ii)}}{\iff} &(x \in \triangle i \iff x \in i_2 \land x \notin i_1)
\end{aligned}$$

and

$$\begin{aligned}
&((x \notin i_1 \lor x \in \triangledown i) \land x \notin \triangle i \iff x \notin i_2) \\
\iff &((x \notin i_1 \lor x \in \triangledown i) \land x \notin \triangle i \land x \in i_1 \iff x \notin i_2 \land x \in i_1) \\
\iff &(x \in \triangledown i \land x \notin \triangle i \land x \in i_1 \iff x \notin i_2 \land x \in i_1) \\
\overset{\text{(ii)}}{\iff} &(x \in \triangledown i \land x \in i_1 \iff x \notin i_2 \land x \in i_1) \\
\overset{\text{(i)}}{\iff} &(x \in \triangledown i \iff x \notin i_2 \land x \in i_1)
\end{aligned}$$

Which is equivalent to:

$$\begin{aligned}
&\triangle i = i_2 - i_1 \\
&\triangledown i = i_1 - i_2
\end{aligned}$$

As this construction always yields exactly one value for $\triangle i$ and $\bigtriangledown i$, and thus for $(\triangle i, \bigtriangledown i)$, the proof is complete. □

Proof of Lemma 3.1 The lemma is proved by proving that selection and projection can always be *pushed up* in the expression tree, and that the application of more than one selection (projection) in sequence can be combined into one selection (projection).

Let D be a database instance, Q, Q_1 and Q_2 be SPJ-expressions, θ and ψ predicates over tuples, and A a set of attribute numbers:

Pushing up a selection through a Cartesian product

$$\begin{aligned}
&x \in D(\sigma_\theta(Q_1) \times Q_2)\\
\iff &\exists q_1, q_2 : x = q_1 \cdot q_2 \wedge q_1 \in D(Q_1) \wedge \theta(q_1) \wedge q_2 \in D(Q_2)\\
\iff &\exists q_1, q_2 : x = q_1 \cdot q_2 \wedge q_1 \in D(Q_1) \wedge \theta(q_1 \cdot q_2) \wedge q_2 \in D(Q_2)\\
\iff &x \in D(\sigma_\theta(Q_1 \times Q_2))
\end{aligned}$$

Pushing up a selection that is the right operand of a Cartesian product can be done analogously.

Pushing up a projection through a Cartesian product

$$\begin{aligned}
&x \in D(\pi_A(Q_1) \times Q_2)\\
\iff &\exists q_1, q_2 : x = q_1|_A \cdot q_2 \wedge q_1 \in D(Q_1) \wedge q_2 \in D(Q_2)\\
\iff &\exists q_1, q_2 : x = (q_1 \cdot q_2)|_{A \cup \{|A|+1,\ldots,|A|+\mathrm{ar}(Q_2)\}} \wedge q_1 \in D(Q_1) \wedge q_2 \in D(Q_2)\\
\iff &x \in D(\pi_{A \cup \{|A|+1,\ldots,|A|+\mathrm{ar}(Q_2)\}}(Q_1 \times Q_2))
\end{aligned}$$

Pushing up a projection that is the right operand of a Cartesian product can be done analogously.

Pushing up a projection through a selection Define θ' as the predicate that does the same as θ, but for tuples before the projection was applied:

$$\theta'(t) = \theta(t|_A)$$

Then:

$$\begin{aligned}
&x \in D(\sigma_\theta(\pi_A(Q)))\\
\iff &x \in D(\pi_A(Q)) \wedge \theta(x)\\
\iff &\exists q : x = q|_A \wedge q \in D(Q) \wedge \theta(x)\\
\iff &\exists q : x = q|_A \wedge q \in D(Q) \wedge \theta(q|_A)\\
\iff &\exists q : x = q|_A \wedge q \in D(Q) \wedge \theta'(q)\\
\iff &\exists q : x = q|_A \wedge x \in D(\sigma_{\theta'}(Q))\\
\iff &x \in D(\pi_A(\sigma_{\theta'}(Q)))
\end{aligned}$$

Combining two consecutive selections

$$\begin{aligned}
& x \in D(\sigma_\theta(\sigma_\psi(Q))) \\
\iff & x \in D(Q) \wedge \theta(x) \wedge \psi(x) \\
\iff & x \in D(Q) \wedge (\theta \wedge \psi)(x) \\
\iff & x \in D(\sigma_{\theta\wedge\psi}(Q))
\end{aligned}$$

Combining two consecutive projections Let B be a set of attribute numbers with $B \subseteq \{1, \ldots, |A|\}$. Let C be constructed as follows:

$$\begin{aligned}
& A = \{a_1, \ldots, a_n\}, \text{ where } a_1 < \cdots < a_n \\
& B = \{b_1, \ldots, b_m\}, \text{ where } b_1 < \cdots < b_m \\
& C = \{a_{b_1}, \ldots, a_{b_m}\}
\end{aligned}$$

Then:

$$\begin{aligned}
& x \in D(\pi_B(\pi_A(Q))) \\
\iff & \exists s : x = s|_B \wedge s \in D(\pi_A(Q)) \\
\iff & \exists q, s : x = s|_B \wedge s = q|_A \wedge q \in D(Q) \\
\iff & \exists q : x = q|_A|_B \wedge q \in D(Q) \\
\iff & \exists q : x = q|_C \wedge q \in D(Q) \\
\iff & x \in D(\pi_C(Q))
\end{aligned}$$

The desired form can now be obtained by starting with the original SPJ-expression, pushing up all selections and projections (where all selections must stay on top of, or are pushed above the projections), combining all selections to one selection, and combining all projections to one projection. $\Box$

Proof of Lemma 4.3 The following holds for every value of x:

$$\begin{aligned}
& (x \in D_{\text{new}}(Q) \iff x \in D_{\text{ev}}((Q - \triangledown Q) \cup \triangle Q)) \\
& \qquad \wedge\, x \notin D_{\text{ev}}(\triangledown Q - Q) \wedge x \notin D_{\text{ev}}(\triangle Q \cap Q) \\
\iff & (x \in D_{\text{new}}(Q) \iff (x \in D_{\text{ev}}(Q) \wedge x \notin D_{\text{ev}}(\triangledown Q)) \vee x \in D_{\text{ev}}(\triangle Q)) \\
& \qquad \wedge\, (x \notin D_{\text{ev}}(\triangledown Q) \vee x \in D_{\text{ev}}(Q)) \\
& \qquad \wedge\, (x \notin D_{\text{ev}}(\triangle Q) \vee x \notin D_{\text{ev}}(Q)) \qquad (1) \\
\overset{\text{Table A.1}}{\iff} & (x \in D_{\text{ev}}(\triangle Q) \iff x \notin D_{\text{ev}}(Q) \wedge x \in D_{\text{new}}(Q)) \\
& \qquad \wedge\, (x \in D_{\text{ev}}(\triangledown Q) \iff x \in D_{\text{ev}}(Q) \wedge x \notin D_{\text{new}}(Q)) \qquad (2)
\end{aligned}$$

The equivalence between (1) and (2) is proved by means of the boolean table in table A.1. $\Box$

Proof of Theorem 4.4 The proof is by induction on the structure of expression Q. Let D_{ev} be $D_{\text{old}} \cup \text{deltas}(t)$. For each subexpression, the following should hold:

$$\begin{aligned}
& D_{\text{new}}(Q) = D_{\text{ev}}((Q - \triangledown(Q)) \cup \triangle(Q)) \\
& D_{\text{ev}}(\triangledown(Q) - Q) = \emptyset \\
& D_{\text{ev}}(\triangle(Q) \cap Q) = \emptyset
\end{aligned}$$

$x \in D_{\text{ev}}(Q)$	$x \in D_{\text{ev}}(\triangle Q)$	$x \in D_{\text{ev}}(\bigtriangledown Q)$	$x \in D_{\text{new}}(Q)$	(1)	(2)
0	0	0	0	1	1
0	0	0	1	0	0
0	0	1	0	0	0
0	0	1	1	0	0
0	1	0	0	0	0
0	1	0	1	1	1
0	1	1	0	0	0
0	1	1	1	0	0
1	0	0	0	0	0
1	0	0	1	1	1
1	0	1	0	1	1
1	0	1	1	0	0
1	1	0	0	0	0
1	1	0	1	0	0
1	1	1	0	0	0
1	1	1	1	0	0

Table A.1: Boolean table that proves the equivalence in the proof of lemma 4.3.

The first property expresses that the result of evaluating $(\triangle(Q), \bigtriangledown(Q))$ is indeed a delta between $D_{\text{old}}(Q)$ and $D_{\text{new}}(Q)$, when evaluated in the context of D_{ev}. The second and third express that this delta is minimal.

Let $\triangle Q$ and $\bigtriangledown Q$ be abbreviations for $\triangle(Q)$ and $\bigtriangledown(Q)$, respectively. According to lemma 4.3, the previous properties are equivalent to: $\forall x$:

$$\begin{aligned} x \in D_{\text{ev}}(\triangle Q) &\iff x \notin D_{\text{ev}}(Q) \wedge x \in D_{\text{new}}(Q) \\ x \in D_{\text{ev}}(\bigtriangledown Q) &\iff x \in D_{\text{ev}}(Q) \wedge x \notin D_{\text{new}}(Q) \end{aligned}$$

These properties are proved by induction on the structure of Q, assuming that they hold for the subexpressions. The first two cases are proved directly using the properties given before.

Q^m is used as an abbreviation for $(Q - \bigtriangledown Q) \cup \triangle Q$. It is clear that $D_{\text{ev}}(Q^m) = D_{\text{new}}(Q)$.

Relation instance Clearly, a constant relation instance is not influenced by changes to the base relations. The proof is trivial:

$$\begin{aligned} &D_{\text{new}}(i) \overset{def}{=} i \overset{def}{=} D_{\text{old}}(i) \\ &\emptyset - i \equiv \emptyset \\ &\emptyset \cap i \equiv \emptyset \end{aligned}$$

Base relation This follows directly from the definitions of transaction and minimality of transactions.

$$\begin{aligned} &D_{\text{new}}(R) \overset{def}{=} D_{\text{old}}((R - \bigtriangledown R) \cup \triangle R) \\ &\bigtriangledown R - R \overset{def}{\equiv} \emptyset \\ &\triangle R \cap R \overset{def}{\equiv} \emptyset \end{aligned}$$

Selection Intuitively, it is clear that selection can just be pushed through. Tuples added to Q that pass the condition should be added to V, and passing tuples deleted from Q should be deleted from V.

$$\begin{aligned}
& x \in D_{\text{ev}}(\triangle(\sigma_\theta(Q))) \\
\iff & x \in D_{\text{ev}}(\sigma_\theta(\triangle Q)) \\
\iff & x \in D_{\text{ev}}(\triangle Q) \wedge \theta(x) \\
\iff & x \notin D_{\text{ev}}(Q) \wedge x \in D_{\text{new}}(Q) \wedge \theta(x) \\
\iff & x \notin D_{\text{ev}}(\sigma_\theta(Q)) \wedge x \in D_{\text{new}}(\sigma_\theta(Q))
\end{aligned}$$

$$\begin{aligned}
& x \in D_{\text{ev}}(\triangledown(\sigma_\theta(Q)) \\
\iff & x \in D_{\text{ev}}(\sigma_\theta(\triangledown Q)) \\
\iff & x \in D_{\text{ev}}(\triangledown Q) \wedge \theta(x) \\
\iff & x \in D_{\text{ev}}(Q) \wedge x \notin D_{\text{new}}(Q) \wedge \theta(x) \\
\iff & x \in D_{\text{ev}}(\sigma_\theta(Q)) \wedge x \notin D_{\text{new}}(\sigma_\theta(Q))
\end{aligned}$$

Projection A projected tuple should be added if it is not already in the result.

$$\begin{aligned}
& x \in D_{\text{ev}}(\triangle(\pi_A(Q))) \\
\iff & x \in D_{\text{ev}}(\pi_A(\triangle Q) - \pi_A(Q)) \\
\iff & x \in D_{\text{ev}}(\pi_A(\triangle Q)) \wedge x \notin D_{\text{ev}}(\pi_A(Q)) \\
\iff & \exists y : y|_A = x \wedge y \in D_{\text{ev}}(\triangle Q) \wedge x \notin D_{\text{ev}}(\pi_A(Q)) \\
\iff & \exists y : y|_A = x \wedge y \notin D_{\text{ev}}(Q) \wedge y \in D_{\text{new}}(Q) \wedge x \notin D_{\text{ev}}(\pi_A(Q)) \\
\iff & x \notin D_{\text{ev}}(\pi_A(Q)) \wedge x \in D_{\text{new}}(\pi_A(Q))
\end{aligned}$$

A projected tuple should be deleted if there are no tuples left that project to the same value.

$$\begin{aligned}
& x \in D_{\text{ev}}(\triangledown(\pi_A(Q))) \\
\iff & x \in D_{\text{ev}}(\pi_A(\triangledown Q) - \pi_A(Q^m) \\
\iff & x \in D_{\text{ev}}(\pi_A(\triangledown Q)) \wedge x \notin D_{\text{ev}}(\pi_A(Q^m) \\
\iff & \exists y : y|_A = x \wedge y \in D_{\text{ev}}(\triangledown Q) \wedge x \notin D_{\text{new}}(\pi_A(Q)) \\
\iff & \exists y : y|_A = x \wedge y \in D_{\text{ev}}(Q) \wedge y \notin D_{\text{new}}(Q) \wedge x \notin D_{\text{new}}(\pi_A(Q)) \\
\iff & x \in D_{\text{ev}}(\pi_A(Q)) \wedge x \notin D_{\text{new}}(\pi_A(Q))
\end{aligned}$$

Union A tuple should be added if it is added to one operand, and was not in the other one before the transaction.

$$\begin{aligned}
& x \in D_{\text{ev}}(\triangle(Q_1 \cup Q_2)) \\
\iff & x \in D_{\text{ev}}((\triangle Q_1 - Q_2) \cup (\triangle Q_2 - Q_1)) \\
\iff & (x \in D_{\text{ev}}(\triangle Q_1) \wedge x \notin D_{\text{ev}}(Q_2)) \vee (x \in D_{\text{ev}}(\triangle Q_2) \wedge x \notin D_{\text{ev}}(Q_1)) \\
\iff & (x \notin D_{\text{ev}}(Q_1) \wedge x \in D_{\text{new}}(Q_1) \wedge x \notin D_{\text{ev}}(Q_2)) \\
& \quad \vee (x \notin D_{\text{ev}}(Q_2) \wedge x \in D_{\text{new}}(Q_2) \wedge x \notin D_{\text{ev}}(Q_1)) \\
\iff & x \notin D_{\text{ev}}(Q_1) \wedge x \notin D_{\text{ev}}(Q_2) \wedge (x \in D_{\text{new}}(Q_1) \vee x \in D_{\text{new}}(Q_2)) \\
\iff & x \notin D_{\text{ev}}(Q_1 \cup Q_2) \wedge x \in D_{\text{new}}(Q_1 \cup Q_2)
\end{aligned}$$

A tuple should be deleted if it is deleted from one operand, and is not in the other one after the transaction.

$$\begin{aligned}
& x \in D_{\text{ev}}(\triangledown(Q_1 \cup Q_2)) \\
\iff & x \in D_{\text{ev}}((\triangledown Q_1 - Q_2^m) \cup (\triangledown Q_2 - Q_1^m)) \\
\iff & (x \in D_{\text{ev}}(\triangledown Q_1) \wedge x \notin D_{\text{ev}}(Q_2^m)) \vee (x \in D_{\text{ev}}(\triangledown Q_2) \wedge x \notin D_{\text{ev}}(Q_1^m)) \\
\iff & (x \in D_{\text{ev}}(Q_1) \wedge x \notin D_{\text{new}}(Q_1) \wedge x \notin D_{\text{new}}(Q_2)) \\
& \quad \vee (x \in D_{\text{ev}}(Q_2) \wedge x \notin D_{\text{new}}(Q_2) \wedge x \notin D_{\text{new}}(Q_1)) \\
\iff & (x \in D_{\text{ev}}(Q_1) \vee x \in D_{\text{ev}}(Q_2)) \wedge x \notin D_{\text{new}}(Q_1) \wedge x \notin D_{\text{new}}(Q_2) \\
\iff & x \in D_{\text{ev}}(Q_1 \cup Q_2) \wedge x \notin D_{\text{new}}(Q_1 \cup Q_2)
\end{aligned}$$

Intersection A tuple should be added if it is added to one operand, and is in the other one after the transaction.

$$\begin{aligned}
& x \in D_{\text{ev}}(\triangle(Q_1 \cap Q_2)) \\
\iff & x \in D_{\text{ev}}((\triangle Q_1 \cap Q_2^m) \cup (\triangle Q_2 \cap Q_1^m)) \\
\iff & (x \in D_{\text{ev}}(\triangle Q_1) \wedge x \in D_{\text{ev}}(Q_2^m)) \vee (x \in D_{\text{ev}}(\triangle Q_2) \wedge x \in D_{\text{ev}}(Q_1^m)) \\
\iff & (x \notin D_{\text{ev}}(Q_1) \wedge x \in D_{\text{new}}(Q_1) \wedge x \in D_{\text{new}}(Q_2)) \\
& \quad \vee (x \notin D_{\text{ev}}(Q_2) \wedge x \in D_{\text{new}}(Q_2) \wedge x \in D_{\text{new}}(Q_1)) \\
\iff & (x \notin D_{\text{ev}}(Q_1) \vee x \notin D_{\text{ev}}(Q_2)) \wedge x \in D_{\text{new}}(Q_1) \wedge x \in D_{\text{new}}(Q_2) \\
\iff & x \notin D_{\text{ev}}(Q_1 \cap Q_2) \wedge x \in D_{\text{new}}(Q_1 \cap Q_2)
\end{aligned}$$

A tuple should be deleted if it is deleted from one operand, and it was in the other one before the transaction.

$$\begin{aligned}
& x \in D_{\text{ev}}(\triangledown(Q_1 \cap Q_2)) \\
\iff & x \in D_{\text{ev}}((\triangledown Q_1 \cap Q_2) \cup (\triangledown Q_2 \cap Q_1)) \\
\iff & (x \in D_{\text{ev}}(\triangledown Q_1) \wedge x \in D_{\text{ev}}(Q_2)) \vee (x \in D_{\text{ev}}(\triangledown Q_2) \wedge x \in D_{\text{ev}}(Q_1)) \\
\iff & (x \in D_{\text{ev}}(Q_1) \wedge x \notin D_{\text{new}}(Q_1) \wedge x \in D_{\text{ev}}(Q_2)) \\
& \quad \vee (x \in D_{\text{ev}}(Q_2) \wedge x \notin D_{\text{new}}(Q_2) \wedge x \in D_{\text{ev}}(Q_1)) \\
\iff & x \in D_{\text{ev}}(Q_1) \wedge x \in D_{\text{ev}}(Q_2) \wedge (x \notin D_{\text{new}}(Q_1) \vee x \notin D_{\text{new}}(Q_2)) \\
\iff & x \in D_{\text{ev}}(Q_1 \cap Q_2) \wedge x \notin D_{\text{new}}(Q_1 \cap Q_2)
\end{aligned}$$

Difference A tuple should be added if it is added to the first operand, and is not in the second operand after the transaction. Also, a tuple should be added if it is deleted from the second operand, and is in the first operand after the transaction (the usage of $(Q_1 - \triangledown Q_1)$ instead of Q_1^m is an optimization).

$$
\begin{aligned}
& x \in D_{\mathrm{ev}}(\triangle(Q_1 - Q_2)) \\
\iff & x \in D_{\mathrm{ev}}((\triangle Q_1 - Q_2^m) \cup (\bigtriangledown Q_2 \cap (Q_1 - \bigtriangledown Q_1))) \\
\iff & (x \in D_{\mathrm{ev}}(\triangle Q_1) \wedge x \notin D_{\mathrm{ev}}(Q_2^m)) \\
& \quad \vee (x \in D_{\mathrm{ev}}(\bigtriangledown Q_2) \wedge x \in D_{\mathrm{ev}}(Q_1) \wedge x \notin D_{\mathrm{ev}}(\bigtriangledown Q_1)) \\
\iff & (x \notin D_{\mathrm{ev}}(Q_1) \wedge x \in D_{\mathrm{new}}(Q_1) \wedge x \notin D_{\mathrm{new}}(Q_2)) \vee (x \in D_{\mathrm{ev}}(Q_2) \\
& \quad \wedge x \notin D_{\mathrm{new}}(Q_2) \wedge x \in D_{\mathrm{ev}}(Q_1) \wedge (x \notin D_{\mathrm{ev}}(Q_1) \vee x \in D_{\mathrm{new}}(Q_1))) \\
\iff & (x \notin D_{\mathrm{ev}}(Q_1) \wedge x \in D_{\mathrm{new}}(Q_1) \wedge x \notin D_{\mathrm{new}}(Q_2)) \\
& \quad \vee (x \in D_{\mathrm{ev}}(Q_2) \wedge x \notin D_{\mathrm{new}}(Q_2) \wedge x \in D_{\mathrm{ev}}(Q_1) \wedge x \in D_{\mathrm{new}}(Q_1)) \\
\iff & (x \notin D_{\mathrm{ev}}(Q_1) \vee (x \in D_{\mathrm{ev}}(Q_2) \wedge x \in D_{\mathrm{ev}}(Q_1))) \\
& \quad \wedge x \in D_{\mathrm{new}}(Q_1) \wedge x \notin D_{\mathrm{new}}(Q_2) \\
\iff & (x \notin D_{\mathrm{ev}}(Q_1) \vee x \in D_{\mathrm{ev}}(Q_2)) \wedge x \in D_{\mathrm{new}}(Q_1) \wedge x \notin D_{\mathrm{new}}(Q_2) \\
\iff & x \notin D_{\mathrm{ev}}(Q_1 - Q_2) \wedge x \in D_{\mathrm{new}}(Q_1 - Q_2)
\end{aligned}
$$

A tuple should be deleted if it is deleted from the first operand, and it was not in the second operand before the transaction. Also, a tuple should be deleted if it is added to the second operand, and it was in the first operand before the transaction.

$$
\begin{aligned}
& x \in D_{\mathrm{ev}}(\bigtriangledown(Q_1 - Q_2)) \\
\iff & x \in D_{\mathrm{ev}}((\bigtriangledown Q_1 - Q_2) \cup (\triangle Q_2 \cap Q_1)) \\
\iff & (x \in D_{\mathrm{ev}}(\bigtriangledown Q_1) \wedge x \notin D_{\mathrm{ev}}(Q_2)) \vee (x \in D_{\mathrm{ev}}(\triangle Q_2) \wedge x \in D_{\mathrm{ev}}(Q_1)) \\
\iff & (x \in D_{\mathrm{ev}}(Q_1) \wedge x \notin D_{\mathrm{new}}(Q_1) \wedge x \notin D_{\mathrm{ev}}(Q_2)) \\
& \quad \vee (x \notin D_{\mathrm{ev}}(Q_2) \wedge x \in D_{\mathrm{new}}(Q_2) \wedge x \in D_{\mathrm{ev}}(Q_1)) \\
\iff & x \in D_{\mathrm{ev}}(Q_1) \wedge x \notin D_{\mathrm{ev}}(Q_2) \wedge (x \notin D_{\mathrm{new}}(Q_1) \vee x \in D_{\mathrm{new}}(Q_2)) \\
\iff & x \in D_{\mathrm{ev}}(Q_1 - Q_2) \wedge x \notin D_{\mathrm{new}}(Q_1 - Q_2)
\end{aligned}
$$

Cartesian Product A tuple should be added if its first part is added to the first operand, and the second part is in the second operand after the transaction. Analogously for when its second part is added to the second operand.

Let A_{Q_1} be $\{1, \ldots, \mathrm{ar}(Q_1)\}$, and A_{Q_2} be $\{\mathrm{ar}(Q_1)+1, \ldots, \mathrm{ar}(Q_1)+\mathrm{ar}(Q_2)\}$.

$$
\begin{aligned}
& x \in D_{\mathrm{ev}}(\triangle(Q_1 \times Q_2)) \\
\iff & x \in D_{\mathrm{ev}}((\triangle Q_1 \times Q_2^m) \cup (Q_1^m \times \triangle Q_2)) \\
\iff & (x|_{A_{Q_1}} \in D_{\mathrm{ev}}(\triangle Q_1) \wedge x|_{A_{Q_2}} \in D_{\mathrm{ev}}(Q_2^m)) \\
& \quad \vee (x|_{A_{Q_1}} \in D_{\mathrm{ev}}(Q_1^m) \wedge x|_{A_{Q_2}} \in D_{\mathrm{ev}}(\triangle Q_2)) \\
\iff & (x|_{A_{Q_1}} \notin D_{\mathrm{ev}}(Q_1) \wedge x|_{A_{Q_1}} \in D_{\mathrm{new}}(Q_1) \wedge x|_{A_{Q_2}} \in D_{\mathrm{new}}(Q_2)) \\
& \quad \vee (x|_{A_{Q_1}} \in D_{\mathrm{new}}(Q_1) \wedge x|_{A_{Q_2}} \notin D_{\mathrm{ev}}(Q_2) \wedge x|_{A_{Q_2}} \in D_{\mathrm{new}}(Q_2)) \\
\iff & (x|_{A_{Q_1}} \notin D_{\mathrm{ev}}(Q_1) \vee x|_{A_{Q_2}} \notin D_{\mathrm{ev}}(Q_2)) \\
& \quad \wedge x|_{A_{Q_1}} \in D_{\mathrm{new}}(Q_1) \wedge x|_{A_{Q_2}} \in D_{\mathrm{new}}(Q_2) \\
\iff & x \notin D_{\mathrm{ev}}(Q_1 \times Q_2) \wedge x \in D_{\mathrm{new}}(Q_1 \times Q_2)
\end{aligned}
$$

A tuple should be deleted if its first part is deleted from the first operand, and the second part was in the second operand before the transaction. Analogously for when its second part is deleted from the second operand.

$$\begin{aligned}
& x \in D_{\mathrm{ev}}(\triangledown(Q_1 \times Q_2)) \\
\iff & x \in D_{\mathrm{ev}}((\triangledown Q_1 \times Q_2) \cup (Q_1 \times \triangledown Q_2)) \\
\iff & (x|_{A_{Q_1}} \in D_{\mathrm{ev}}(\triangledown Q_1) \wedge x|_{A_{Q_2}} \in D_{\mathrm{ev}}(Q_2)) \\
& \quad \vee (x|_{A_{Q_1}} \in D_{\mathrm{ev}}(Q_1) \wedge x|_{A_{Q_2}} \in D_{\mathrm{ev}}(\triangledown Q_2)) \\
\iff & (x|_{A_{Q_1}} \in D_{\mathrm{ev}}(Q_1) \wedge x|_{A_{Q_1}} \notin D_{\mathrm{new}}(Q_1) \wedge x|_{A_{Q_2}} \in D_{\mathrm{ev}}(Q_2)) \\
& \quad \vee (x|_{A_{Q_1}} \in D_{\mathrm{ev}}(Q_1) \wedge x|_{A_{Q_2}} \in D_{\mathrm{ev}}(Q_2) \wedge x|_{A_{Q_2}} \notin D_{\mathrm{new}}(Q_2)) \\
\iff & x|_{A_{Q_1}} \in D_{\mathrm{ev}}(Q_1) \wedge x|_{A_{Q_2}} \in D_{\mathrm{ev}}(Q_2) \\
& \quad \wedge (x|_{A_{Q_1}} \notin D_{\mathrm{new}}(Q_1) \vee x|_{A_{Q_2}} \notin D_{\mathrm{new}}(Q_2)) \\
\iff & x \in D_{\mathrm{ev}}(Q_1 \times Q_2) \wedge x \notin D_{\mathrm{new}}(Q_1 \times Q_2)
\end{aligned}$$

This finishes the proof. □

Proof of Theorem 4.7 This proof is by induction on the structure of Q.

Constant relation

$$\lim_{b/d \to \infty} \frac{0}{1} = 0$$

Relation reference

$$\lim_{b/d \to \infty} \frac{d}{b} = 0$$

Selection

$$\lim_{b/d \to \infty} \frac{c_{\mathtt{inc}}(Q', b, d)}{c_{\mathtt{ev}}(Q', b)} \overset{hyp}{=} 0$$

Union, intersection, and difference

$$\lim_{b/d \to \infty} \frac{c_{\mathtt{inc}}(Q_1, b, d) + c_{\mathtt{inc}}(Q_2, b, d)}{c_{\mathtt{ev}}(Q_1, b) + c_{\mathtt{ev}}(Q_2, b)} \overset{hyp}{=} 0$$

Cartesian product

$$\lim_{b/d \to \infty} \frac{(c_{\mathtt{ev}}(Q_1, b) + c_{\mathtt{ev}}(Q_2, b))(c_{\mathtt{inc}}(Q_1, b, d) + c_{\mathtt{inc}}(Q_2, b, d))}{c_{\mathtt{ev}}(Q_1, b) c_{\mathtt{ev}}(Q_2, b)} \overset{hyp}{=} 0$$

This concludes the proof. □

Auteursrechterlijke overeenkomst

Opdat de Universiteit Hasselt uw eindverhandeling wereldwijd kan reproduceren, vertalen en distribueren is uw akkoord voor deze overeenkomst noodzakelijk. Gelieve de tijd te nemen om deze overeenkomst door te nemen, de gevraagde informatie in te vullen (en de overeenkomst te ondertekenen en af te geven).

Ik/wij verlenen het wereldwijde auteursrecht voor de ingediende eindverhandeling:
Incremental Evaluation of Relational Expressions
Richting: **Licentiaat in de informatica** Jaar: **2007**
in alle mogelijke mediaformaten, - bestaande en in de toekomst te ontwikkelen - , aan de Universiteit Hasselt.

Niet tegenstaand deze toekenning van het auteursrecht aan de Universiteit Hasselt behoud ik als auteur het recht om de eindverhandeling, - in zijn geheel of gedeeltelijk -, vrij te reproduceren, (her)publiceren of distribueren zonder de toelating te moeten verkrijgen van de Universiteit Hasselt.

Ik bevestig dat de eindverhandeling mijn origineel werk is, en dat ik het recht heb om de rechten te verlenen die in deze overeenkomst worden beschreven. Ik verklaar tevens dat de eindverhandeling, naar mijn weten, het auteursrecht van anderen niet overtreedt.

Ik verklaar tevens dat ik voor het materiaal in de eindverhandeling dat beschermd wordt door het auteursrecht, de nodige toelatingen heb verkregen zodat ik deze ook aan de Universiteit Hasselt kan overdragen en dat dit duidelijk in de tekst en inhoud van de eindverhandeling werd genotificeerd.

Universiteit Hasselt zal mij als auteur(s) van de eindverhandeling identificeren en zal geen wijzigingen aanbrengen aan de eindverhandeling, uitgezonderd deze toegelaten door deze overeenkomst.

Ik ga akkoord,

Nicolas Barbier

Datum: **11.09.2007**