Ausführungsplanoptimierung in PostgreSQL

Belegarbeit

eingereicht am Fachbereich

Informatik

der Hochschule Zittau/Görlitz (HAW)

als Prüfungsleistung im Fach

Fortgeschrittene Datenbank-Konzepte 2

vorgelegt von:

Christof Ochmann (35989)

Ingo Körner (40586)

Görlitz, 9. Juli 2012

Betreuer: Prof. ten Hagen

Abstract

Diese Arbeit baut auf das Projekt Datenbankkonfigurationen[1], dass während der Vorlesungsreihe ADBC1 erstellt wurde, auf. Im Projekt Datenbankkonfigurationen wird untersucht, wie sich die Ausführungsgeschwindigkeit von Abfragen steigern lässt. Im Projekt Ausführungsplanoptimierung in PostgreSQL wird darüberhinaus untersucht, welche Ausführungspläne, für bestimmte Konfigurationen und bestimmte Queries erzeugt werden. Und wie sich diese Pläne auf die Ausführungsgeschwindigkeit von SQL-Queries auswirken. In beiden Projekten wird nur der Bereich OLAP betrachtet. Diese Arbeit behandelt nur Datenbankkonfigurationen, die Einfluss auf den Ausführungsplan haben. Für alle Konfigurationen die keinen Einfluss haben, wird der Standardwert von PostgreSQL beibehalten. Ziel der Arbeit ist, Annahmen über die Ausführungsgeschwindigkeiten verschiedener Ausführungspläne zu treffen, diese theoretisch zu begründen und dann durch Messergebnisse praktisch zu belegen. Anhand der Messergebnisse werden die aufgestellten Hypothesen bestätigt oder wiederlegt. Für wiederlegte Hypothesen wird eine Begründung gesucht.

[1] https://dl.dropbox.com/u/608146/ADBC1%20OLAP.pdf

Inhaltsverzeichnis

Abbildungsverzeichnis

Listings

Abkürzungsverzeichnis

DBMS	Datenbankmanagementsystem
ERD	Entity-Relationship Diagram
OLAP	Online Analytical Processing
SQL	Structured Query Language

Literaturverzeichnis

[1] Martin, Robert C. (2008): Clean Code: A Handbook of Agile Software Craftsmanship. Prentice Hall International

[2] Freeman, Eric (2007): Entwurfsmuster von Kopf bis Fuß. O'REILLY

[3] `http://www.postgresql.org/ (08.06.2012)`

[4] `http://wiki.postgresql.org/ (08.06.2012)`

1 Theorie

1.1 Einleitung

Ziel dieses Projektes ist zu zeigen, welche Ausführungspläne, für welche Queries bei welchen Indexarten bei welchen Spalten die Ausführungszeit beschleunigen.

Es wird davon ausgegangen, dass die zur Verfügung stehenden Indexarten sowie die Schreibweise des Queries, den Ausführungsplan und damit die Ausführungsgeschwindigkeit verändern können.

1.2 Aufgabenstellung

In diesem Projekt werden Ausführungspläne für bestimmte Queries untersucht. Dazu werden Queries ausgewählt, die im Bereich OLAP und Data Warehouse angesiedelt sind. Als Grundlage wird ein Datenbankentwurf für ein Projekt aus der Vorlesungsreihe XML genommen. Die Tabellen sollen mit Testdaten gefüllt werden. Dazu ist der Datengenerator aus dem Projekt Datenbankkonfigurationen anzupassen. Die Abfragen werden auf den gefüllten Tabellen angewendet. Die Ausführungspläne, die dafür erzeugt werden, werden nach Performancegesichtspunkten untersucht.

1.3 Relevanz des Forschungsgegenstandes

Der Forschungsgegenstand dieser Arbeit ist, Annahmen über erzeugte Ausführungspläne von Abfragen zu treffen und gegebenfalls wiederlegte Annahmen zu erklären. Der Forschungsgegenstand ist relevant, da bisher keine konkreten Aus-

führungspläne für die gewählten Abfragen vorliegen. Ziel dieser Foschung ist es, Ausführungspläne zu finden, die die höchste Ausführungsgeschwindigkeit für alle Abfragen bringt.

Um die optimalen Ausführungspläne zu finden, muss sich vertiefend in eine PostgreSQL eingearbeitet werden. Das geschieht z.B. unter Zuhilfenahme von Büchern und Online-Ressourcen. In diesen Medien ist der Forschungsstand zur Erstellung von Ausführungsplänen dokumentiert. Bei der Anpassung des Datengenerators müssen darüberhinaus technische Probleme gelöst werden.

1.4 Der aktuelle Wissensstand

Noch nicht vorhandene Kenntnisse über die Erstellung von Ausführunsplänen werden haupt- sächlich aus Onlineressourcen bezogen. Primärliteratur zur gewählten Datenbank ist unter www.postgresql.org zu ?nden. Unter dieser Adresse ist das gewählte DBMS dokumentiert. Auf dieser Seite wird eine Einfšuhrung in PostgreSQL gegeben. Es gibt Installationsanleitungen, eine umfassende Dokumentation von PostgreSQL und Werkzeuge die ein e?zienteres Arbeiten mit PostgreSQL ermöglichen.

1.5 PostgreSQL

Da im Projekt Datenbankkonfigurationen bereits viel Erfahrung mit MySQL gesammelt wurde, wird im aktuellen Projekt nun nach einem DBMS gesucht, mit qualitativ ähnlichen Eigenschaften zu MySQL. Ziel ist dabei auch andere DBMS kennen zu lernen. Für die Untersuchung der Ausführungspläne wird dazu PostgreSQL gewählt, ein freies, objektrelationales Datenbankmanagementsystem für das mit PgAdmin auch eine grafische Benutzeroberfläche zur Verfügung steht.

1.6 Was ist ein Ausführungsplan?

In einem Ausführungsplan wird beschrieben, in welchen Schritten ein DBMS einen Query ausführt. Auch die Reihenfolge der Schritte wird dabei angegeben. Da in einem Query nur beschrieben steht, was im Ergebnis gewünscht ist, aber kein Algorithmus, wie dieses Ergebnis errechnet werden soll, gibt es viele verschiedene Möglichkeiten, zu dem Ergebnis zu kommen. Das DBMS soll dabei den Ausführungsplan finden, der das Ergebnis effizient errechnet.

Technisch betrachtet, ist der Ausführungsplan ein Baum. Die Abarbeitung des Ausführungsplans beginnt bei seinen Blättern. In den Blättern des Baumes stehen die Zugriffspfade (access paths). Der Zugriffspfad wird durch den Scan-Algorithmus angegeben. Ein Scan-Algorithmus gibt an, wie eine Tabelle durchlaufen wird.

1.7 Die Abarbeitung von Abfragen in PostgreSQL

1. **Empfang des SQL-Befehls**
   Nachdem der SQL-Befehl über eine Netzwerkverbindung übertragen wurde, findet die Kodierungsumwandlung statt und die weiteren Phasen der Abarbeitung sehen den Befehl in der Servercodierung. Hierbei gibt es nur sehr geringe Optimierungsmöglichkeiten. Es können theoretisch CPU-Zyklen gespart werden, wenn die Clientkodierung gleich der Servercodierung ist, ansonsten wird eine Konvertierung durchgeführt. Diese Auswirkungen sind jedoch sehr gering. Der Parameter *client_encoding* informiert den Server darüber, welche Kodierung die ankommenden Befehle haben und welche Kodierung das Anfrageergebnis haben soll, welches an den Client gesendet wird. Die Voreinstellung gibt an welche Kodierung der Server intern verwendet.

2. **Parser**
   In dieser Abarbeitungsphase wird die kodierte Befehlszeichenkette durch einen internen Parse-Baum dargestellt. Des weiten wird die Befehlszeichenkette auf semantische Bedingungen überprüft und etwas bearbeitet. Die SQL-Befehle werden dann aufgeteilt in sogenannte optimierbare Anweisungen(SELECT, INSERT, UPDATE und DELETE) und Hilfsanweisungen. Die Hilfsanweisungen werden später direkt ausgeführt und sie erzeugen keine

Ausgabe. Dagegen kommen die optimierbaren Anweisungen in den Rewriter. Für den Parser gibt es von der Anwenderseite keine Möglichkeit die Geschwindigkeit zu optimieren.

3. **Query Rewriter**
   Der Rewriter wendet die Anfrageumschreibregeln(Query Rewrite Rules) an. Dabei werden die Sichten(Views) und andere benutzerdefinierte Regeln aufgelöst, in die Anfrage eingebaut und im Parse-Baum ersetzt. Da der Rewriter vor dem Planer angesiedelt ist, bekommt der Planer es nicht mit, ob die Anfrage aus einer Sicht kam oder nicht. Mit der Erstellung einer Sicht hat man somit keinen Optimierungsvorteil.

4. **Planer / Optimizer**
   Der Planer bekommt den möglicherweise umgeschriebenen Parse-Baum und hat die Aufgabe einen Ausführungsplan(execution plan) zu erstellen, der ebenfalls ein Baum ist. Der Ausführungsplan beschreibt wie auf die Tabellen zugegriffen werden soll, also welche Indexe und Join-Algorithmen verwendet werden sollen und in welcher Reihenfolge. Es soll möglichst der optimalste und schnellste Ausführungsplan gefunden werden.

5. **Executor**
   Der vom Planer auserwählte Ausführungsplan wird vom Executor ausgeführt. Dabei werden Zugriffsrechte auf Tabellen und andere Objekte und Constraints geprüft. Die Laufzeit der Ausführung hängt nicht nur davon ab ob der Plan gut ist, sondern auch von der gesamten Systemkonfiguration.

1.8 Ausführungsplan verändern

In PostgreSQL hat der Anwender keine Möglichkeit selbst einen Plan vorzugeben, aber mit bestimmten Parametern kann man den Anfrageplaner beeinflussen und somit den Ausführungsplan verändern. Man kann einzelne Plantypen ausschalten und den Planer dazu bringen nur die aktivierten Plantypen zu verwenden. Die Parameter werden während einer Datenbanksitzung mit dem SET-Befehl gesetzt:

z.B. SET enable_seqscan TO off;

In der Voreinstellung sind alle Plantypen an und können vom Planer benutzt werden. Die folgende Auflistung zeigt die verschiedenen Plantypen und ihre Bedeutung:

- enable_seqscan:
  Wenn der Planer zu sequenziellen Plänen tendiert, kann man mit diesem Parameter die Verwendung von indexbasierten Plänen erzwingen. Die sequenziellen Pläne werden zwar nicht ganz abgeschaltet, aber sie werden extrem hoch bewertet und erscheinen zu aufwendig für den Planer um sie zu nutzen. Auf die selbe Art und Weise können die folgenden Parameter den Planer beeinflussen:

- enable_indexscan:
  Hiermit werden die indexbasierten Pläne aktiviert bzw. deaktiviert.

- enable_bitmapscan:
  Auch das Aktivieren und Deaktivieren von Bitmap Index Scans ist möglich.

- enable_nestloop:
  Aktivierung bzw. Deaktivierung von Nested-Loop-Joins.

- enable_hashjoin:
  Aktivierung bzw. Deaktivierung von Hash-Join-Plantypen.

- enable_mergejoin:
  Aktivierung bzw. Deaktivierung von Merge-Join-Plantypen.

- enable_hashagg:
  Aktivierung bzw. Deaktivierung von Hash-basierter Aggregierung.

- enable_sort:
  Sortieroperationen werden vom Planer nicht berücksichtigt wenn dieser Parameter deaktiviert ist. Es kann aber vorkommen, dass sie trotzdem im Anfrageplan vorkommen wenn keine Alternativen herangezogen werden können.

- enable_tidscan:
  Aktivierung bzw. Deaktivierung von TID-basierten Plantypen.

1.9 Kostenparameter

Außer den Plantypen gibt es noch die Kostenparameter, die den erwarteten I/O und CPU-Aufwand zählen. Ändert man diese Konfigurationsparameter, dann werden die Anfragen andere Plankosten haben und es wird eventuell ein anderer, günstigerer Ausführungsplan gewählt. Man muss die Kostenfaktoren so einstellen, dass der Plan dem Planer auch am schnellsten erscheint und somit ausgewählt wird.

Die Voreinstellungen dieser Parameter stellen das ungefähre Verhalten eines typischen Computers dar und müssen lediglich an das eigene System angepasst werden. Jedoch gibt es kein Benchmark-Programm, mit dem man diese Einstellungen automatisch vornehmen könnte. In der Regel sind keine weitreichenden Änderungen an diesen Parametern notwendig, es sein denn man nutzt diese als Mittel um einen anderen Ausführungsplan zu erzwingen. Die Änderungen sollten in kleinen Schritten und mit ausführlichen Tests für alle in Frage kommenden Anfragen vorgenommen werden, denn eine falsche Konfiguration kann bei bestimmten Anfragen zu einem hohen Ressourcenverbrauch führen. Die wichtigsten Kostenparameter sind:

- seq_page_cost:
  Wird eine Page von der Platte gelesen, entsteht Aufwand, der mit diesem Parameter definiert ist. Er gibt die Kosten für das sequenziellen Lesen einer Page(typischerweise 8kb) während einer laufenden Leseanweisung an, die Voreinstellung für diesen Parameter ist 1.0.

- random_page_cost:
  Der Parameter gibt an, um wie viel aufwändiger es ist, Pages in zufälliger Reihenfolge zu lesen, wie es bei Indexscans vorkommt (gemessen am sequentiellem Lesen von Pages). Die Voreinstellung ist 4. Die Parameter seq_page_cost und random_page_cost kann man sich vereinfacht als Kosten für sequenzielle Scans und Indexscans vorstellen. Indexscans sind langsamer als das sequenzielle Lesen, dafür aber muss man in der Regel bei den Indexscans nicht so viele Seiten lesen. Mit diesen Parametern entscheidet der Planer zwischen Index und nicht Index. Die Voreinstellung besagt, dass ein Indexscan viermal langsamer ist als der sequenzielle Scan. Dieser Wert kann jedoch bei schnellen Festplatten zu hoch sein und es wird empfohlen den Wert pauschal auf 2.0 oder 3.0 herunterzusetzen. Wenn sich die Datenbank komplett im

Hauptspeicher befindet, ist es sinnvoll die beiden Parameter gleichzusetzen, da der Aufwand in diesem Fall gleich ist.

- cpu_tuple_cost:
  Hiermit werden die Kosten für das Verarbeiten einer Tabellenzeile durch die CPU definiert. Die Voreinstellung ist 0.01 und dieser Wert wird in der Regel sehr selten verändert.

- cpu_index_tuple_cost:
  Dieser Parameter definiert die Kosten für das Verarbeiten eines Indexeintrags durch die CPU während eines Indexscans. Auch dieser Parameter wird sehr selten verändert und ist auf 0.005 voreingestellt.

- cpu_operator_cost:
  Dieser Konfigurationsparameter gibt die Kosten für das Ausführen eines Operators oder einer Funktion an. Die Voreinstellung ist 0.0025 und wird in der Regel sehr selten verändert.

- effective_cache_size:
  Dieser Parameter gibt an, wie viel Cache-Speicher für eine Anfrage vom Betriebssystem bereitgestellt werden. Es ist nur eine Vermutung und es wird kein Speicher angelegt. Der Planer sieht anhand dieses Parameters ob ein Index im Hauptspeicher höchstwahrscheinlich gecached ist, oder noch von der Festplatte gelesen werden muss. Daraufhin entscheidet der Planer ob ein indexbasierter Zugriffspfad zu wählen ist oder ob nicht eine andere Alternative kostengünstiger ist. Es hoher Wert sorgt also dafür, dass mehr Indexscans verwendet werden. Die Voreinstellung beträgt 128 MByte.

1.10 Indexe

Ein Index auf eine Spalte lohnt sich nur bei hoher Selektivität. Wenn die Selektivität nicht hoch ist, muss sowieso die gesamte Tabelle durchgegangen werden, d.h. jeder Block angefasst werden. Wenn der Planer mit analyze genug Statistiken hat, entscheidet er, ob er den Index verwendet oder nicht.

Ein Index lohnt sich nicht, wenn man viele Ergebnisse erwartet und einen nur die ersten zehn Zeilen interessieren. Denn dann kann sich der Planer das durchhashen

und sortieren ersparen. Der Planer sollte den Index nicht verwenden, wenn ein Limit von z.B. zehn angegeben wurde.

1.10.1 Indextypen

Von PostgreSQL werden derzeit vier Indextypen unterstützt:

- B-tree:
  Es ist die Implementierung des B+-Baums und der Standardindextyp. Er kann alle Anfragen mit den Vergleichsoperatoren(¡, ¡=, =, ¿=, ¿) und den Konstrukten wie BETWEEN und IN bearbeiten.

- Hash:
  Dieser Indextyp verwendet eine Hashtabelle und bedient nur Anfragen mit dem Gleichheitsoperator(=). Er bietet keine bessere Geschwindigkeit als B-tree und wird heutzutage hauptsächlich nur bei Experimenten verwendet.

- GiST:
  Bei GiST handelt es sich um eine universelle Schnittstelle, um verschiedene anwendungsspezifische Indextypen selbst definieren zu können. Zwei Anwendungen davon sind PostGIS und OpenFTS(Open Source Full Textj Search). Man verwendet diesen Indextyp um beispielsweise Geodaten zu sortieren oder bei der Volltextsuche.

- GIN:
  Hierbei handelt es sich um einen invertierten Index, der Werte mit mehreren Schlüsseln wie beispielsweise Arrays und Listen indizieren kann.

1.10.2 Pläne mit Bitmap Index Scan

Der reine Indexscan eignet sich am besten dort, wo die Selektivität sehr hoch ist und der reine Sequential Scan sollte bei niedriger Selektivität verwendet werde, also dann wenn die Tabelle komplett oder fast komplett gelesen werden soll. Der Bitmap Index Scan dagegen soll das gesamte Spektrum zwischen Index- und Sequential Scan abdecken, also die Vorteile von beiden Indextypen kombinieren.

```
EXPLAIN Select *
from public.user u
where u.gender = 'w'
AND u.birthday between '01.01.1986' AND '31.12.2013'
```

Query Plan:

```
"Bitmap Heap Scan on "user" u  (cost=64.62..236.66 rows=1412 width=61)"
"  Recheck Cond: ((birthday >= '1986-01-01'::date)
AND (birthday <= '2013-12-31'::date))"
"  Filter: (gender = 'w'::text)"
"  ->  Bitmap Index Scan on user_birthday
(cost=0.00..64.27 rows=2802 width=0)"
"        Index Cond: ((birthday >= '1986-01-01'::date)
AND (birthday <= '2013-12-31'::date))"
```

Wie man bei diesem Ausführungsplan sehen kann, besteht der Bitmap Index Scan aus zwei Schritten. Zuerst werden alle Treffer aus dem Index ermittelt und in einer Bitmap im RAM zwischengespeichert und sortiert. Anschließend werden die Treffer beim Bitmap Heap Scan in der Tabelle der Reihe nach aufgesucht. Die Sprünge zwischen Tabelle und Index werden somit verhindert. Der Bitmap Index Scan liefert dem Bitmap Heap Scan die Eingabe und ist ihm somit untergeordnet, was man bei einem Ausführungsplan an dem Pfeil erkennen kann. Die Startkosten für den Bitmap Heap Scan sind größer null, d.h., dass der Bitmap Index Scan abgeschlossen sein muss, bevor der Bitmap Heap Scan seine Arbeit beginnen kann.

1.11 Plananalyse

Mit dem Befehl EXPLAIN gefolgt von der eigentlichen Anfrage können die Ausführungpläne angesehen werden:

Beispiel1 ohne Indexe:

```
EXPLAIN Select u.userId, u.name, u.email, u.gender, u.birthday
from public.user u, public.event e, public.participation p
```

```
where e.eventname = 'event1'
AND e.eventid = p.eventid
AND p.userid = u.userid;

"Hash Join  (cost=480.53..506.29 rows=1 width=38)"
"  Hash Cond: (u.userid = p.userid)"
"  ->  Seq Scan on "user" u  (cost=0.00..22.00 rows=1000 width=38)"
"  ->  Hash  (cost=480.52..480.52 rows=1 width=8)"
"        ->  Hash Join  (cost=279.01..480.52 rows=1 width=8)"
"              Hash Cond: (p.eventid = e.eventid)"
"              ->  Seq Scan on participation p
(cost=0.00..164.00 rows=10000 width=16)"
"              ->  Hash  (cost=279.00..279.00 rows=1 width=8)"
"                    ->  Seq Scan on event e
(cost=0.00..279.00 rows=1 width=8)"
"                          Filter: (eventname = 'event1'::text)"
```

- Die erste Zahl nach dem „cost“ sind die Startkosten. Es ist der geschätzte Aufwand, den der Executor investieren muss, bevor der Planknoten Ergebnisse produzieren kann. Eine 0.00 bedeutet, dass bei einem sequenziellen Scan die Ergebnisse sofort ausgegeben werden, sobald der Scan ausgeführt wird.

- Die zweite Zahl ist der geschätzte Aufwand für die Abarbeitung des Planknotens und somit die interessante von den beiden Zahlen.

- Die Zeilenzahl(rows) ist eine Schätzung über die Anzahl der ausgegebenen Ergebniszeilen. Anhand dieser Zahl wird die Abwägung getroffen, ob ein Indexscan günstiger ist oder auch nicht und ob andere Planvarianten infrage kommen.

- Die letzte Zahl gibt die Größe der Ergebniszeile in Bytes an(width). Man kann mit Hilfe der Zeilenzahl(rows) den Speicherbedarf der Ergebnismenge vorhersehen(rows * width).

Die geschätzten Kosten für einen sequenziellen Scan der Tabelle „user“, die in unserem Beispiel1 auf 22.00 geschätzt werden, lassen sich mit Hilfe folgender Anfrage

in PostgreSQL leicht berechnen:

SELECT relpages, reltuples FROM pg_class WHERE relname = 'user'

Das Ergebnis dieser Anfrage ist:
relpages: 12
reltuples: 1000

Das bedeutet, dass die Tabelle „user“ aus 12 Diskpages besteht und 1000 Zeilen enthält.

Der geschätzte Aufwand für die Abarbeitung des Planknotens, der nur aus einem sequenziellen Scan besteht (Seq Scan on „user“ u (cost=0.00..22.00 rows=1000 width=38)), wird wie folgt berechnet:

(Anzahl der Diskpages(je 8 KByte) x seq_page_cost) + (rows x cpu_tuple_cost))
In unserem Beispiel1: (12 * 1.0) + (1000 * 0.01) = 22

Also die Anzahl der Seiten mal die Kosten für das sequenzielle Lesen einer Page plus die Anzahl der Zeilen mal die Kosten für die Abarbeitung einer Zeile in der CPU.

1.12 Planvergleich

MIt dem Befehl **EXPLAIN ANALYZE** kann man die geschätzten Kosten mit den Ergebnissen der Ausführung vergleichen. Die Anfrage wird auch auch tatsächlich ausgeführt:

```
"Hash Join  (cost=480.53..506.29 rows=1 width=38)
(actual time=19.767..20.204 rows=1 loops=1)"
"  Hash Cond: (u.userid = p.userid)"
"  ->  Seq Scan on "user" u
(cost=0.00..22.00 rows=1000 width=38)
(actual time=0.029..0.279 rows=1000 loops=1)"
"  ->  Hash  (cost=480.52..480.52 rows=1 width=8)
```

```
(actual time=19.439..19.439 rows=1 loops=1)"
"        Buckets: 1024  Batches: 1  Memory Usage: 1kB"
"        ->  Hash Join  (cost=279.01..480.52 rows=1 width=8)
(actual time=9.220..19.428 rows=1 loops=1)"
"              Hash Cond: (p.eventid = e.eventid)"
"              ->  Seq Scan on participation p
(cost=0.00..164.00 rows=10000 width=16)
(actual time=0.037..4.178 rows=10000 loops=1)"
"              ->  Hash  (cost=279.00..279.00 rows=1 width=8)
(actual time=9.109..9.109 rows=1 loops=1)"
"                    Buckets: 1024  Batches: 1  Memory Usage: 1kB"
"                    ->  Seq Scan on event e
(cost=0.00..279.00 rows=1 width=8)
(actual time=0.040..9.102 rows=1 loops=1)"
"                          Filter: (eventname = 'event1'::text)"
"Total runtime: 20.386 ms"
```

Die Zahlen in der ersten Klammer sind dieselben wie bei **EXPLAIN** und die Daten in der zweiten Klammer enthalten analog zu den Kosten die tatsächliche Ausführungszeit sowie die tatsächliche Anzahl der Zeilen. Der Parameter loops zeigt an wie oft dieser Teilplan ausgeführt wird, spielt jedoch erst bei Joins eine Rolle, denn bei einfachen Anfragen ist die Zahl logischerweise eine 1.

Bei **EXPLAIN ANALYZE** ist es am wichtigsten darauf zu achten, ob die Zeilenschätzung richtig war. Bei größeren Abweichungen müssen die Statistiken verbessert werden.

1.13 Statistiken

Der Planer sammelt Statistiken um die Anzahl der zu erwartenden Ergebniszeilen zu ermitteln. Relevant dabei ist die Anzahl der Zeilen in der Tabelle, sowie die Anzahl der von der Tabelle belegten Blöcke. Diese Daten stehen in der Systemtabelle pg_class und der Planer kann mit Hilfe dieser Informationen die Anzahl der Zeilen für einen sequenziellen Scan über eine Tabelle, sowie den Speicher- und

I/O-Aufwand ermitteln. Aktuell gehalten werden die Statistiken von den Befehlen **VACUUM** und **ANALYZE**, sowie von **CREATE INDEX**. Meistens jedoch werden die Tabellen nicht komplett gelesen, sondern müssen nach bestimmten Kriterien z.B. mit der WHERE-Klausel ausgewählt werden. Dazu muss der Planer weitere Statistiken sammeln, die in der Tabelle pg_statistics stehen und mit der folgenden Anfrage eingesehen werden können:

SELECT * FROM pg_stats WHERE tablename = 'user' AND attname = 'birthday'

Das Ergebnis dieser Anfragen sind Näherungswerte und beziehen sich immer nur auf eine Spalte. Die interessantesten sind:

- n_distinct = 80: Die Zahl bedeutet, dass die Tabelle 80 unterschiedliche Werte enthält.
- most_common_vals = 1951-02-01, 1947-02-01, usw.: Ein Array mit den häufigsten Werten in der Tabelle.
- correlation = 0.016471: Eine Korrelation zwischen der logischen Reihenfolge der Werte und der Reihenfolge auf der Festplatte. Ein Wert der nah bei -1 oder +1 liegt bedeutet, dass Indexscans günstiger sind, weil die Festplattenzugriffe nicht so weit auseinander liegen.

1.14 Die drei Scan-Algorithmen

Ein Scan-Algorithmus arbeitet immer nur auf einer einzelnen Tabelle. In PostgreSQL gibt es die folgenden drei Scan-Algorithmen:

1. sequential scan (full table scan)
   Der Inhalt der Tabelle wird komplett gelesen. Er wird blockweise vom Sekundärspeicher wie z.B. einer Festplatte in den Arbeitsspeicher geholt.
2. index scan
   Hat eine Tabelle einen Index, kann er verwendet werden, um die Tupel sor-

tiert zu lesen. Bei einem Index Scan werden Blöcke auch mehrmals gelesen, wenn der Inhalt der Tabelle nicht auch sortiert in den Blöcken vorliegt. Das ist relativ teuer und nur für kleine Treffermengen geeignet. Ein Index-Scan eignet sich bei einer hohen Selektivität eines Selects.

3. bitmap index scan
   Hier wird der Index gescannt und ein Bitmap mit den getroffenen Blocknummern erzeugt. Das Bitmap der Blocknummern wird dann aufsteigend sortiert. Die Tabelle wird anhand der sortierten Bitmap-Blocknummern aufsteigend gescannt. Das ist nur möglich, wenn Indexe für die betreffenden Spalten existieren.

1.15 Die drei Join-Algorithmen

In den Knoten des Ausführungsplanes stehen Datenbankoperatoren wie Projektion, Selektion, Kreuzprodukt, Vereinigung, Differenz oder Umbenennung.

Die Hintereinanderausführung der Operationen kartesisches Produkt und Selektion wird Join genannt. Joins werden im DBMS intern über Join-Algorithmen realisiert. Join-Algorithmen verknüpfen Tabellen paarweise miteinander.

In PostgreSQL gibt es drei grundlegende JOIN-Algorithmen:

1. nested loop join
   Für jede Zeile aus der treibenden Tabelle wird die innere Tabelle einmal durchlaufen. Wenn die innere Tabelle indiziert ist, kann sie mit einem Index-Scan durchlaufen werden. Ein Nested Loop Join kann sehr kostenintensiv werden, wenn er die innere Tabelle mehrmals lesen muss. Wenn auf der inneren Tabelle ein Index-Scan erfolgen kann, oder die innere Tabelle sehr klein ist, kann sich diese durch vorherige Anfragen im Cache befinden und so schnell abgearbeitet werden. Wenn es sich um einen sequentiellen Scan handelt, der alle Zeilen vergleicht, dann muss die innere Tabelle u.U. so oft von der Festplatte gelesen werden, wie die treibende Tabelle Zeilen hat. Wird nur die erste übereinstimmende Zeile gesucht, ist der Nested Loop Join schneller als andere Joins, die vorher erst ihr komplettes Ergebnis berechnen müssen, bevor sie den ersten Treffer zurückgeben können. Der Nested Loop Join ver-

langt vor dem Query keinerlei Investition, wie Hashing oder Sortierung.

2. hash join
   Für einen Hashjoin müssen beide Tabellen als Hash-Tabelle vorliegen. Das setzt vorraus, dass bevor ein Hash Join eingesetzt werden kann, die Tabellen durchgehasht wurden, d.h. ein Hashwert für das spätere Join-Attribut gebildet wird. Wie der Nested Loop Join ist der Hash-Join besonders performant, wenn der Größenunterschied zwischen treibender Tabelle und innerer Tabelle groß ist und die kleinere Tabelle komplett in den Speicher passt. Dazu wird bei der Ausführung des Hash-Joins die kleinere Hashtabelle in den Arbeitsspeicher geladen, mit dem JOIN-Attribut als Schlüssel. Dann wird die größere Tabelle gescannt und jeder gefundene Wert wird als Schlüssel für die kleinere Hashtable benutzt. Ein Hash-Join kann nur dann verwendet werden, wenn die Spalten mit dem = Operator verglichen werden. Der Performancezugewinn des Hash-Join wird durch einen höheren Sekundärspeicherbedarf erkauft, denn die Hashtabellen werden im Tempspace materialisiert. Bei einem Hash Join muss immer ein Materialize erfolgen, der die Buckets erzeugt. Der Hash-Join eignet sich auch dann, wenn alle Lösungen gebraucht werden und nicht nur z.B. die ersten zehn.

3. merge join
   Bevor ein Merge Join ausgeführt werden kann, müssen beide Tabellen nach dem Join-Attributen sortiert werden. Liegen beide Tabellen sortiert vor, werden bei einem Merge Join beide Tabellen parallel gescannt und passende Zeilen werden zusammengefügt. Sowohl die treibende als auch die innere Tabelle muss nur einmal gescannt werden. Die vorrangegangene Sortierung erfolgt in einem extra Sortierschritt oder durch die Verwendung eines Index, falls das Feld indiziert ist und der JOIN über dieses Attribut erfolgt. Die Sortierung im ersten Fall ist teurer als wenn ein entsprechender Index vorhanden ist. Werden mehrere Merge-JOINS über dasselbe Attribut hintereinander ausgeführt, muss nur einmal sortiert werden. Wie bei dem Hash-Join wird der Performancezugewinn des Merge Joins mit einem höheren Sekundärspeicherbedarf erkauft, denn beide Tabellen müssen sortiert im Tempspace vorliegen. Der Merge Join ist wie der Hash-Join vor allem dann interessant, wenn alle Lösungen gefunden werden sollen. Anders als bei Nested Loop Join und Hash-Join spielt bei einem Merge Join der Größenunterschied der beiden zu joinenden Tabellen für die Ausführungsgeschwindigkeit keine Rolle.

2 Umsetzung

2.1 Datenbankentwurf

In Abbildung 2.1 auf Seite 17 ist der verwendete Datenbankentwurf zu sehen. Auf ihm werden die zu entwickelnden Queries gefahren.

2.2 Der Datengenerator

Über den grundsätzlichen Aufbau des Datengenerators wird im Projekt Datanbankkonfigurationen[1] eingegangen.

Um die Daten schneller in die Tabellen einzufügen, werden im Datengenerator die generierten Testdaten anders als in DB-Writer nicht mehr über Prepared Statements in die Tabellen eingefügt, sondern über die write-Methode von java.io.Writer in eine Datei geschrieben. Um das umzusetzen, wurde die Komponente DB-Writer durch eine Writer-Komponente ersetzt.

Da sich auch das Data Model in diesem Projekt geändert hat, müssen weitere Komponenten des Generators angepasst werden. Die angepassten Komponenten des Generators zeigt Abbildung 2.2 auf Seite 18.

Die Daten werden im CSV-Format in die jeweilige Datei geschrieben, um sie mit dem Copy-Befehl[2] von PostgreSQL in die Tabelle laden zu können.

Listing 2.1: COPY

```
1 COPY public.User (userId, name, email, gender, birthday,
    password, image) From 'C:\User.txt' DELIMITER ';'
```

[1] https://dl.dropbox.com/u/608146/ADBC1%20OLAP.pdf
[2] http://www.pgadmin.org/docs/1.4/pg/sql-copy.html

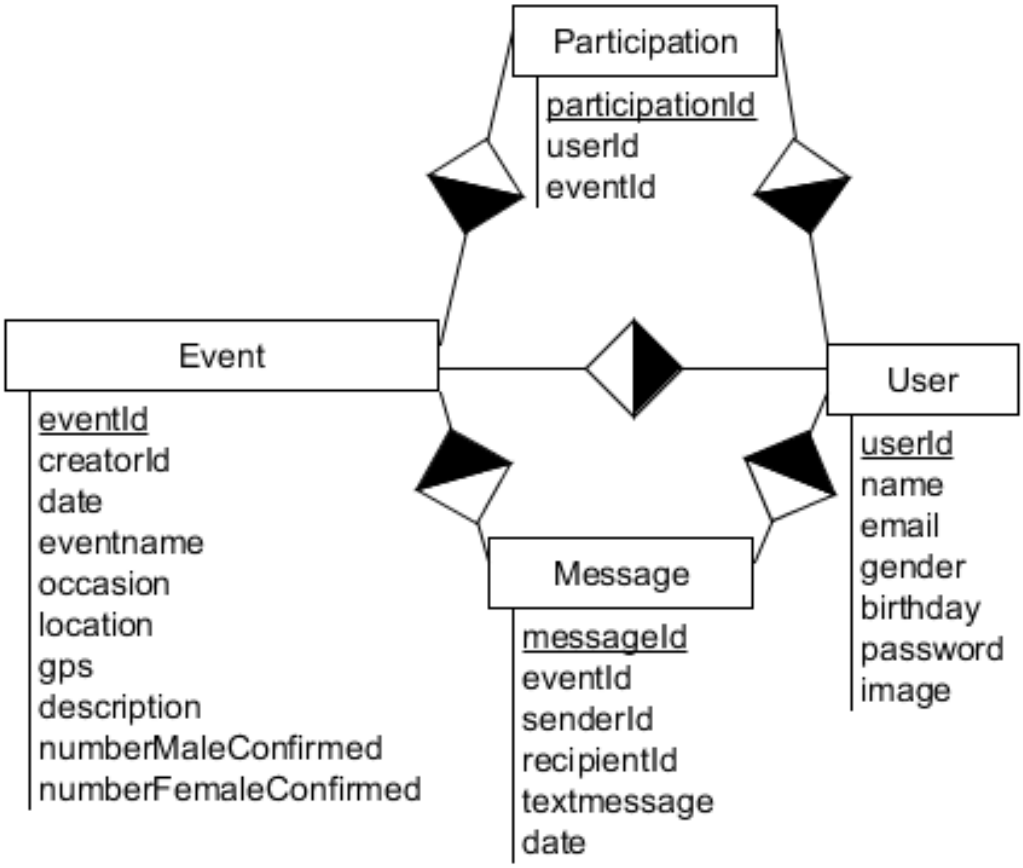

Abbildung 2.1: EER-Diagramm

Dadurch ergibt sich bei einem Umfang von 3,1 Mio generierter Zeilen im Schnitt eine Zeitersparnis um den Faktor zehn - 66 Sekunden für die Generierung und den Import der CSV-Datei in die Tabellen mit COPY, zu 687 Sekunden mit prepared statements.

Das Projekt liegt als Maven-Eclipse-Projekt unter:

https://github.com/rinkdotrink/ComeTogether.git

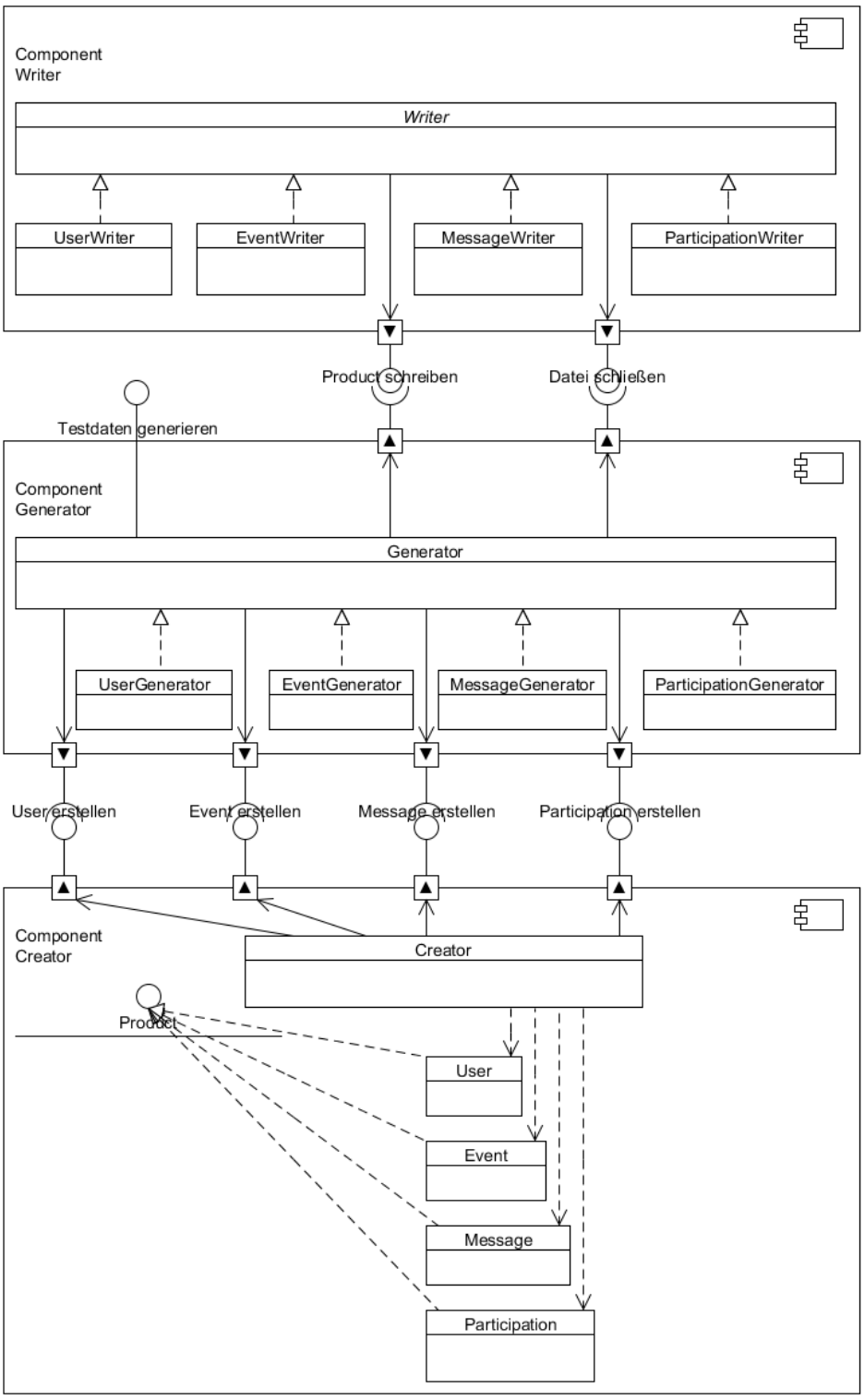

Abbildung 2.2: Komponentendiagramm Datengenerator

2.3 Datenbankabfragen

Query 1: Alle user anzeigen, die am event mit dem eventnamen „event1“ teilnehmen.

Listing 2.2: Query 1

```
1 Select u.userId, u.name, u.email, u.gender, u.birthday
2 from public.user u, public.event e, public.participation p
3 where e.eventname = 'event1'
4 AND e.eventid = p.eventid
5 AND p.userid = u.userid;
```

Query 2: Alle weiblichen user anzeigen, die zwischen 1986 und 1992 geboren worden und an einem event teilnehmen, dass zwischen dem 01.01.2013 und dem 01.03.2013 stattfindet, und bei dem numberMaleConfirmed / numberFemaleConfirmed kleiner 0,5 ist.

Listing 2.3: Query 2

```
1 Select u.userId, u.name, u.email, u.gender, u.birthday
2 from public.user u, public.event e, public.participation p
3 where u.gender = 'w'
4 AND u.birthday between '01.01.1986' AND '31.12.1992'
5 AND e.date between '01.01.2013' AND '01.03.2013'
6 AND e.eventid = p.eventid
7 AND p.userid = u.userid
8 AND e.numberMaleConfirmed / e.numberFemaleConfirmed < 0.5;
```

Query 3: Die Anzahl der textmessages gruppiert und absteigend sortiert nach numberFemaleConfirmed, die zwischen dem 01.01.2010 und dem 31.12.2012 geschrieben wurden, in denen das Wort Salsa vorkommt, deren Empfänger männlich sind und zwischen 1972 und 1982 geboren wurden, deren Absender weiblich sind und die zwischen 1986 und 1992 geboren worden und an einem event teilnehmen, dass zwischen dem 01.01.2013 und dem 01.03.2013 in einem 100km Radius zu der GPS-Koordinate 11.5833 45.1500 stattfindet und bei dem numberMaleConfirmed / numberFemaleConfirmed kleiner 0,5 ist. Das Ergebnis soll nur die ersten fünf Treffer liefern.

Listing 2.4: Query 3

```
1  Select e.numberFemaleConfirmed, Count(m.messageid) as
       anzahlMessages
2  from public.user u, public.event e, public.participation p,
       public.message m
3  where u.gender = 'w'
4  AND u.birthday between '01.01.1986' AND '31.12.1992'
5  AND e.date between '01.01.2013' AND '01.03.2013'
6  AND m.date between '01.01.2010' AND '31.12.2012'
7  AND m.textmessage like '%Salsa'
8  AND e.eventid = p.eventid
9  AND p.userid = u.userid
10 AND m.senderId = u.userId
11 AND e.numberMaleConfirmed / e.numberFemaleConfirmed < 0.5
12 AND DEGREES(acos(cos(RADIANS(90-e.lat))*cos(RADIANS
       (90-11.5833))+sin(RADIANS(90-e.lat))*
13 sin(RADIANS(90-11.5833))*cos(RADIANS(e.lon-45.1500))))
       /360*40074 < 100
14 AND m.recipientId in
15 (Select u2.userId
16 from public.user u2
17 where u2.gender = 'm'
18 AND u2.birthday between '01.01.1972' AND '31.12.1982'
19 )
20 Group by e.numberFemaleConfirmed
21 Order by e.numberFemaleConfirmed DESC
22 Limit 5
```

2.4 Was ist ein Hints-System?

In anderen DBMS wie DB2 oder Oracle kann der Optimizer durch Hinweise (hints) dazu gebracht werden, seinen Ausführungsplan zu verändern.

Der Query in Listing 2.5 auf Seite 21 gibt in MySQL dem Optimizer den Hinweis,

die Tabellen so miteinander zu verknüpfen, wie sie definiert wurden: Es wird tabA als treibende Tabelle und tabB als innere Tabelle verwendet.

Listing 2.5: STRAIGHT JOIN

```
1 SELECT STRAIGHT_JOIN *
2 FROM tabA a, tabB b
3 WHERE a.id = b.id;
```

Wenn MySQL den falschen Index aus einer Menge von möglichen Indexen nimmt, kann z.B. wie in Listing 2.6 dem Optimizer der Hinweis gegeben werden, nur die im folgenden angegebenden Indexe für die Abfrage zu verwenden.

Listing 2.6: USE INDEX

```
1 SELECT * FROM table1 USE INDEX (col1_index, col2_index)
2   WHERE col1=1 AND col2=2 AND col3=3;
```

Mit dem Hinweis IGNORE INDEX (*col2_index*) in Listing 2.7 wird der Optimizer veranlasst, den Index *col3_index* für die Abfrage nicht zu verwenden.

Listing 2.7: IGNORE INDEX

```
1 SELECT * FROM table1 IGNORE INDEX (col3_index)
2   WHERE col1=1 AND col2=2 AND col3=3;
```

In PostgreSQL gibt es kein hints-System[3]. Es sind in einem Query keine Konstrukte vorgesehen, dem Planner Hinweise zu geben, wie er den Ausführungsplan erstellen soll.

Überholte Hints - wenn z.B. der Hint für die aktuelle Tabellengrößen ungeeignet ist - oder die fehlerhafte Anwendung von Hints - können zu suboptimalen Ausführungsplänen führen und somit die Ausführungsgeschwindigkeit negativ beeinflussen. Die Fehlerquelle „menschliches Versagen“ beim Schreiben von Hints wird durch den Verzicht auf ein Hints-System reduziert.

Ohne ein Hints-System wird es allerdings auch schwieriger den Planer verschiedene Ausführungspläne erzeugen zu lassen. Und der Planer ist mehr auf sich allein gestellt, der er keine Hilfe von menschlicher Seite in Form von Hints erwarten kann.

[3] http://wiki.postgresql.org/wiki/OptimizerHintsDiscussion

Auch wenn der Schreiber des Queries sich auf den Planer verlassen muss, sollte er trotzdem ein paar Dinge beachten, auf die im Folgenden eingegangen wird.

2.5 Planerverwirrung

Wie können Ausführungspläne noch beschleunigt werden? In PostgreSQL durch Umschreiben nur sehr begrenzt. Es ist zwar möglich, die Queries auf verschiedene Weisen zu schreiben, aber die Schreibweise hat auf den Optimizer nur in seltenen Fällen Einfluss. Wie sehen diese Fälle aus? Es gibt Querykonstruktionen[4], die den Planner verwirren können und somit zu langsamen Ausführungsplänen führen würden.

Wird der Operator > oder der Operator < verwendet, kann der Planer die Tabellen nicht mehr mit Hilfe eines Hash-JOIN verknüpfen:

Listing 2.8: kein Hash Join möglich

```
1 Select u.userId, p.eventid
2 from public.user u, public.event e, public.participation p
3 where e.eventid < p.eventid + 1
4 AND e.eventid > p.eventid - 1
5 AND p.userid < u.userid + 1
6 AND p.userid > u.userid - 1;
```

Listing 2.9: Ausführungsplan nested loop join

```
1 "Nested Loop  (cost=0.00..283418724402242.31 rows
    =1192507407395482 width=16)"
2 "  Join Filter: ((e.eventid < (p.eventid + 1)) AND (e.
    eventid > (p.eventid - 1)))"
3 "  -> Nested Loop  (cost=0.00..2641025223.00 rows
    =11111111111 width=16)"
4 "        Join Filter: ((p.userid < (u.userid + 1)) AND (p.
    userid > (u.userid - 1)))"
5 "        -> Seq Scan on participation p  (cost
    =0.00..22739.00 rows=1000000 width=16)"
7
```

[4] http://www.postgresql.org/docs/current/interactive/explicit-joins.html

```
6  "          -> Materialize  (cost=0.00..3125.00 rows=100000
       width=8)"
8  "                -> Seq Scan on "user" u  (cost
       =0.00..2234.00 rows=100000 width=8)"
9  "  -> Materialize  (cost=0.00..52708.96 rows=965931 width
       =8)"
10 "          -> Seq Scan on event e  (cost=0.00..44105.31 rows
       =965931 width=8)"
```

besser:

Listing 2.10: Hash Join möglich

```
1 explain Select u.userId, p.eventid
2 from public.user u, public.event e, public.participation p
3 where e.eventid = p.eventid
4 AND p.userid = u.userid;
```

Listing 2.11: Ausführungsplan hash join

```
1  "Hash Join  (cost=43997.00..146580.72 rows=965931 width=16)
       "
2  "  Hash Cond: (p.userid = u.userid)"
3  "  -> Hash Join  (cost=40122.00..113562.10 rows=965931
       width=16)"
4  "          Hash Cond: (e.eventid = p.eventid)"
5  "          -> Seq Scan on event e  (cost=0.00..44105.31 rows
       =965931 width=8)"
6  "          -> Hash  (cost=22739.00..22739.00 rows=1000000
       width=16)"
7  "                -> Seq Scan on participation p  (cost
       =0.00..22739.00 rows=1000000 width=16)"
8  "  -> Hash  (cost=2234.00..2234.00 rows=100000 width=8)"
9  "          -> Seq Scan on "user" u  (cost=0.00..2234.00 rows
       =100000 width=8)"
```

Bei Tabellenspalten, die auf Gleichheit geprüft werden, kann der Planer sich für einen Hash-Join entscheiden. Wenn beide Operanden in dem Vergleich Ergebnis einer Berechnung sind, ist der Planer gezwungen, auf einen Merge-Join zurückzugreifen.

Listing 2.12: merge join

```
1 explain Select u.userId, p.eventid
2 from public.user u, public.event e, public.participation p
3 where (e.eventid + 1) - 1  = (p.eventid + 1) - 1
4 AND (p.userid + 1) - 1 = (u.userid + 1) -1
```

Listing 2.13: Ausführungsplan merge join

```
1 "Merge Join  (cost=136351958.69..60508303947.65 rows
    =2414827500000 width=16)"
2 "  Merge Cond: ((((e.eventid + 1) - 1)) = (((p.eventid + 1)
     - 1)))"
3 "  -> Sort  (cost=166544.39..168959.22 rows=965931 width
    =8)"
4 "        Sort Key: (((e.eventid + 1) - 1))"
5 "        -> Seq Scan on event e  (cost=0.00..44105.31 rows
    =965931 width=8)"
6 "  -> Materialize  (cost=136185414.30..138685414.30 rows
    =500000000 width=16)"
7 "        -> Sort  (cost=136185414.30..137435414.30 rows
    =500000000 width=16)"
8 "              Sort Key: (((p.eventid + 1) - 1))"
9 "              -> Merge Join  (cost=168485.16..12672485.16
     rows=500000000 width=16)"
10 "                    Merge Cond: ((((u.userid + 1) - 1)) =
    (((p.userid + 1) - 1)))"
11 "                    -> Sort  (cost=11907.32..12157.32
    rows=100000 width=8)"
12 "                          Sort Key: (((u.userid + 1) - 1))
    "
13 "                          -> Seq Scan on "user" u  (cost
    =0.00..2234.00 rows=100000 width=8)"
14 "                    -> Materialize  (cost
    =156577.84..161577.84 rows=1000000 width=16)"
15 "                          -> Sort  (cost
    =156577.84..159077.84 rows=1000000 width=16)"
16 "                                Sort Key: (((p.userid + 1)
```

```
    - 1))"
17 "                                    -> Seq Scan on
    participation p  (cost=0.00..22739.00 rows=1000000 width
    =16)"
```

Wenn nur ein Operand in dem Vergleich Ergebnis einer Berechnung ist, greift der Planner auf einen nested loop Join zurück:

Listing 2.14: nested loop join

```
1 explain Select u.userId, p.eventid
2 from public.user u, public.event e, public.participation p
3 where (e.eventid + 1) - 1  = 5
4 AND (p.userid + 1) - 1 = 7
```

Listing 2.15: Ausführungsplan nested loop join

```
1 "Nested Loop  (cost=0.00..30193621347.37 rows=2415000000000
    width=16)"
2 "  -> Nested Loop  (cost=0.00..6091095.87 rows=483000000
    width=8)"
3 "        -> Seq Scan on "user" u  (cost=0.00..2234.00 rows
    =100000 width=8)"
4 "        -> Materialize  (cost=0.00..51373.94 rows=4830
    width=0)"
5 "              -> Seq Scan on event e  (cost
    =0.00..51349.79 rows=4830 width=0)"
6 "                    Filter: (((eventid + 1) - 1) = 5)"
7 "  -> Materialize  (cost=0.00..30264.00 rows=5000 width=8)
    "
8 "        -> Seq Scan on participation p  (cost
    =0.00..30239.00 rows=5000 width=8)"
9 "              Filter: (((userid + 1) - 1) = 7)"
```

Es kann gesagt werden, dass Operanden in einem Vergleich nicht das Ergebnis einer Berechnung sein sollten.

2.6 Reihenfolge von Joins erzwingen

Um den Planner zu der angegebenen Join-Reihenfolge zu zwingen, kann das

join_collapse_limit[5] auf 1 gesetzt werden.

Um den Planner zu zwingen, Subqueries nicht in einen JOIN umzuandeln, kann das *from_collapse_limit* auf 1 gesetzt werden.

Der Standardwert für *join_collapse_limit* und *from_collapse_limit* ist acht. Bei z.B. zwölf zu verknüpfenden Tabellen wird keine vollständige Suche mehr nach der besten Joinreihenfolge ausgeführt, sondern eine wahrscheinlichkeitstheoretische genetische Suche die nur noch eine begrenzte Zahl von möglichen Joinreihenfolgen betrachtet. Die genetische Suche braucht weniger Zeit als die vollständige Suche, findet aber nicht zwangsläufig die bestmögliche Joinreihenfolge. Ab welchem Schwellwert die genetische Suche aktiv wird, kann bei *geqo_threshold* gesetzt werden. Der Standardwert ist zwölf.

Auch wenn *join_collapse_limit* auf den Wert eins gesetzt wird, wird bei folgendem Select die JOIN-Reihenfolge vom Planer bestimmt:

Listing 2.16: Reihenfolge vom Planer bestimmt

```
1 SELECT * FROM a, b, c WHERE a.id = b.id AND b.ref = c.id;
```

Erst wenn zwei Tabellen ausdrücklich mit dem Wort JOIN verknüpft werden, zwingt das den Planner, diese zwei Tabellen in der gegebenen Reihenfolge zu verknüpfen:

Listing 2.17: Reihenfolge vom Entwickler bestimmt

```
1 SELECT * FROM a CROSS JOIN b CROSS JOIN c WHERE a.id = b.id
      AND b.ref = c.id;
2 SELECT * FROM a JOIN (b JOIN c ON (b.ref = c.id)) ON (a.id
    = b.id);
```

[5] http://www.postgresql.org/docs/current/interactive/explicit-joins.html

2.7 Ausführungspläne für Queries mit und ohne Index

Query 1: Alle user anzeigen, die am event mit dem eventnamen „event1“ teilnehmen.

Listing 2.18: Query 1

```
1 Select u.userId, u.name, u.email, u.gender, u.birthday
2 from public.user u, public.event e, public.participation p
3 where e.eventname = 'event1'
4 AND e.eventid = p.eventid
5 AND p.userid = u.userid;
```

Listing 2.19: Ausführungsplan Query 1

```
1 "Hash Join  (cost=254.19..619.78 rows=3595 width=108)"
2 "  Hash Cond: (p.userid = u.userid)"
3 "  -> Hash Join  (cost=231.39..467.41 rows=1498 width=8)"
4 "        Hash Cond: (p.eventid = e.eventid)"
5 "        -> Seq Scan on participation p  (cost
    =0.00..160.64 rows=9664 width=16)"
6 "        -> Hash  (cost=231.00..231.00 rows=31 width=8)"
7 "              -> Seq Scan on event e  (cost=0.00..231.00
    rows=31 width=8)"
8 "                    Filter: (eventname = 'event1'::text)"
9 "  -> Hash  (cost=16.80..16.80 rows=480 width=108)"
10 "        -> Seq Scan on "user" u  (cost=0.00..16.80 rows
    =480 width=108)"
```

Da bisher auf keine Spalte ein Index gelegt wurde, werden die Daten aus den Tabellen mit einem Sequential Scan geholt. Um die Tabellen event und participation zu joinen wird ein Hash-Join verwendet, da das Join-Attribute eventid der beiden Tabellen mit dem Gleichheitsoperator verglichen wird. Bei diesem Ausführungsplan ist zu sehen, dass Hash Joins nichts mit Hash Indexen zu tun haben, die in diesem Beispiel gar nicht existieren. Ein Hash Join nutzt keinen Hash-Index sondern eine temporäre Hashtabelle, die on-the-fly für den Join erstellt wird. Die Ergebnisrelation die entsteht, nachdem die Event- und die Participationtabelle gejoint wurden

wird nun wiederum mit der Tabelle User gejoint. Da auch hier das Join-Attribut userid der beiden Relationen mit dem Gleichheitsoperator verglichen wird, kommt wieder ein Hash-Join zum Einsatz.

Die geschätzten Kosten für die Ausführung werden in Disk Page Fetches gemessen. Die erste Zahl steht für die Kosten, die entstehen, bevor die erste Zeile zurückgegeben werden kann. Die zweite Zahl steht für die Kosten, die entstehen, wenn alle Zeilen zurückgegeben werden. In dem Beispiel wird deutlich, dass für einen Sequential Scan keine vorherige Investition nötig ist und somit auch keine Kosten entstehen, bevor die erste Zeile geliefert werden kann.

Beim sequentiellen Scannen der Events vom Datenträger verursacht das Filtern des Events mit dem Namen 'event1' keine zusätzlichen Kosten, da dafür keine zusätzlichen Disk Pages vom Datenträger geholt werden müssen. Es werden nur alle verworfen, die nicht den Namen event1 haben.

Der Hash-Join der Tabellen Event und Participation verlangt, dass die Tabelle Event komplett durchgehasht wird. Um sie durchhashen zu können, muss der Sequential Scan komplett abgeschlossen sein. D.h. die Startkosten für das Durchhashen der Tabelle Event sind mindestens so groß, wie die Gesamtkosten für das vorherige Lesen vom Datenträger.

Der Hash-Join der Tabellen Event und Participation hat mindestens die Startkosten der Gesamtkosten des Durchhashens der Tabelle Event.

Nun werden Primär- und Fremdschlüssel für alle Tabellen vergeben:

Listing 2.20: Primär- und Fremdschlüssel setzen

```
1 ALTER TABLE public.event ADD PRIMARY KEY (eventid);
2 ALTER TABLE public.message ADD PRIMARY KEY (messageid);
3 ALTER TABLE public.participation ADD PRIMARY KEY (
      participationid);
4 ALTER TABLE public.user ADD PRIMARY KEY (userid);
5 ALTER TABLE event ADD CONSTRAINT event_creatorid FOREIGN
      KEY (creatorid) REFERENCES public.user (userid) MATCH
      FULL;
6 ALTER TABLE message ADD CONSTRAINT message_eventid FOREIGN
      KEY (eventid) REFERENCES event (eventid) MATCH FULL;
```

```
7  ALTER TABLE message ADD CONSTRAINT message_senderid FOREIGN
       KEY (senderid) REFERENCES public.user (userid) MATCH
       FULL;
8  ALTER TABLE message ADD CONSTRAINT message_recipientid
       FOREIGN KEY (recipientid) REFERENCES public.user (userid
       ) MATCH FULL;
9  ALTER TABLE participation ADD CONSTRAINT
       participation_userid FOREIGN KEY (userid) REFERENCES
       public.user (userid) MATCH FULL;
10 ALTER TABLE participation ADD CONSTRAINT
       participation_eventid FOREIGN KEY (eventid) REFERENCES
       event (eventid) MATCH FULL;
```

Listing 2.21: Ausführungsplan Query 1 mit Primär- und Fremdschlüsseln

```
1 "Nested Loop  (cost=279.63..495.98 rows=50 width=108)"
2 "  -> Hash Join  (cost=279.63..481.63 rows=50 width=8)"
3 "        Hash Cond: (p.eventid = e.eventid)"
4 "        -> Seq Scan on participation p  (cost
      =0.00..164.00 rows=10000 width=16)"
5 "        -> Hash  (cost=279.00..279.00 rows=50 width=8)"
6 "              -> Seq Scan on event e  (cost=0.00..279.00
      rows=50 width=8)"
7 "                    Filter: (eventname = 'event1'::text)"
8 "  -> Index Scan using user_pkey on "user" u  (cost
      =0.00..0.27 rows=1 width=108)"
9 "        Index Cond: (userid = p.userid)"
```

Nach der Vergabe der Primär- und Fremdschlüssel wird ein anderer Ausführungsplan erzeugt. Es fällt auf, dass die Tabelle user nicht mehr über einen Sequential Scan eingelesen wird, sondern über einen Index Scan. Das wird verursacht weil PostgreSQL beim Anlegen von Primärschlüsseln auch gleich einen Index für jedes Primärschlüsselattribut anlegt. Und da die Relation, die durch den Hash-Join von Event und Participation entsteht per nested loop auf userid = p.userid abfragt, kann p.userid als Schlüssel des Index Scan verwendet werden. PostgreSQL weiß, da es sich um einen Primärschlüssel handelt, wird auch nur höchstens eine Zeile zurückgeliefert. Und da die referentielle Integrität es verbietet, dass in

einem Fremdschlüsselattribut ein Wert steht, der als Primärschlüssel nicht vorkommt, wird auch mindestens eine Zeile geliefert. Der Planer schätzt, dass 50 Zeilen mit dem Eventnamen event1 geliefert werden. Und es werden 10000 Zeilen aus der Tabelle Participation geliefert. Werden diese beiden Tabellen über die Join-Bedingung p.eventid = e.eventid verknüpft, bleiben 50 Zeilen übrig. Der nested loop join hat die äußere Tabelle mit den vorraussichtlich 50 Tupeln. Die Kosten für den nested loop ergeben sich aus der Anzahl der Tupel der äußeren Tabelle multipliziert mit den Kosten der inneren Tabelle (50 * 0.27 = 13,5) plus ein wenig CPU-Zeit für die Join-Verarbeitung. Werden von den Gesamtkosten für das nested loop die Gesamtkosten vom Hash-Join abgezogen, so ergeben sich Kosten von 495.98 - 481.63 = 14.35. Die CPU-Zeit für die Join-Verarbeitung sind 14,35 - 13,5 = 0,85.

Jetzt werden zusätzlich noch Indexe für die Fremdschlüssel angelegt:

Listing 2.22: Indexe für alle Fremdschlüssel setzen

```
1 CREATE INDEX event_creatorid ON public.event(creatorid);
2 CREATE INDEX message_eventid ON public.message(eventid);
3 CREATE INDEX message_senderid ON public.message(senderid);
4 CREATE INDEX message_recipientid ON public.message(
    recipientid);
5 CREATE INDEX participation_userid ON public.participation(
    userid);
6 CREATE INDEX participation_eventid ON public.participation(
    eventid);
```

Listing 2.23: Ausführungsplan Query 1 mit Index für alle Fremdschlüssel

```
1 "Nested Loop  (cost=0.00..287.57 rows=1 width=38)"
2 "  -> Nested Loop  (cost=0.00..287.28 rows=1 width=8)"
3 "        -> Seq Scan on event e  (cost=0.00..279.00 rows=1
     width=8)"
4 "              Filter: (eventname = 'event1'::text)"
5 "        -> Index Scan using participation_eventid on
    participation p  (cost=0.00..8.27 rows=1 width=16)"
6 "              Index Cond: (eventid = e.eventid)"
7 "  -> Index Scan using user_pkey on "user" u  (cost
    =0.00..0.27 rows=1 width=38)"
```

```
8 "          Index Cond: (userid = p.userid)"
```

An diesem Ausführungsplan wird deutlich, dass jetzt für alle gefundenen Events mit dem Namen event1 ihr Attribut e.eventid als Schlüssel für den Index Scan auf Participation verwendet werden kann. Denn die Tabelle Participation besitzt jetzt einen Index auf den Fremdschlüssel eventid. Die Tabelle event wird dagegen sequentiell gescannt, da hier die komplette Tabelle nach dem Event mit dem Namen event1 durchsucht werden muss.

An dieser Stellt sollen noch Indexe auf Nicht-Id-Spalten vergeben werden:

Listing 2.24: Indexe für alle Fremdschlüssel setzen

```
1  CREATE INDEX event_date ON public.event(date);
2  CREATE INDEX event_eventname ON public.event(eventname);
3  CREATE INDEX event_occasion ON public.event(occasion);
4  CREATE INDEX event_location ON public.event(location);
5  CREATE INDEX event_lon ON public.event(lon);
6  CREATE INDEX event_lat ON public.event(lat);
7  CREATE INDEX event_numbermaleconfirmed ON public.event(
       numbermaleconfirmed);
8  CREATE INDEX event_numberfemaleconfirmed ON public.event(
       numberfemaleconfirmed);
9  CREATE INDEX message_textmessage ON public.message(
       textmessage);
10 CREATE INDEX message_date ON public.message(date);
11 CREATE INDEX user_name ON public.user(name);
12 CREATE INDEX user_email ON public.user(email);
13 CREATE INDEX user_gender ON public.user(gender);
14 CREATE INDEX user_birthday ON public.user(birthday);
```

Listing 2.25: Ausführungsplan Query 1 mit Indexen für alle Fremdschlüssel

```
1 "Nested Loop  (cost=0.00..16.84 rows=1 width=38)"
2 "  ->  Nested Loop  (cost=0.00..16.55 rows=1 width=8)"
3 "        ->  Index Scan using event_eventname on event e  (
      cost=0.00..8.27 rows=1 width=8)"
4 "              Index Cond: (eventname = 'event1'::text)"
```

```
5 "        -> Index Scan using participation_eventid on
    participation p  (cost=0.00..8.27 rows=1 width=16)"
6 "              Index Cond: (eventid = e.eventid)"
7 "  -> Index Scan using user_pkey on "user" u  (cost
    =0.00..0.27 rows=1 width=38)"
8 "        Index Cond: (userid = p.userid)"
```

Der nun erzeugte Ausführungsplan senkt die Gesamtkosten von vorher 287.57 auf jetzt 16.84. Erreicht wurde diese Geschwindigkeitssteigerung durch den Index auf die Spalte eventname.

Query 2: Alle weiblichen user anzeigen, die zwischen 1986 und 1992 geboren worden und an einem event teilnehmen, dass zwischen dem 01.01.2013 und dem 01.03.2013 stattfindet, und bei dem numberMaleConfirmed / numberFemaleConfirmed kleiner 0,5 ist.

Listing 2.26: Query 2

```
1 Select u.userId, u.name, u.email, u.gender, u.birthday
2 from public.user u, public.event e, public.participation p
3 where u.gender = 'w'
4 AND u.birthday between '01.01.1986' AND '31.12.1992'
5 AND e.date between '01.01.2013' AND '01.03.2013'
6 AND e.eventid = p.eventid
7 AND p.userid = u.userid
8 AND e.numberMaleConfirmed / e.numberFemaleConfirmed < 0.5;
```

Listing 2.27: Ausführungsplan für Query 2

```
1 "Hash Join  (cost=218.37..511.08 rows=2 width=108)"
2 "  Hash Cond: (e.eventid = p.eventid)"
3 "  -> Seq Scan on event e  (cost=0.00..292.60 rows=10
    width=8)"
4 "        Filter: ((date >= '2013-01-01'::date) AND (date <=
     '2013-03-01'::date) AND (((numbermaleconfirmed /
    numberfemaleconfirmed))::numeric < 0.5))"
5 "  -> Hash  (cost=217.77..217.77 rows=48 width=116)"
```

```
6  "        -> Hash Join  (cost=20.41..217.77 rows=48 width
       =116)"
7  "              Hash Cond: (p.userid = u.userid)"
8  "              -> Seq Scan on participation p  (cost
       =0.00..160.64 rows=9664 width=16)"
9  "              -> Hash  (cost=20.40..20.40 rows=1 width
       =108)"
10 "                    -> Seq Scan on "user" u  (cost
       =0.00..20.40 rows=1 width=108)"
11 "                          Filter: ((birthday >=
       '1986-01-01'::date) AND (birthday <= '1992-12-31'::date)
        AND (gender = 'w'::text))"
```

mit Primär- und Fremdschlüsseln für alle Tabellen:

Listing 2.28: Ausführungsplan Query 2 mit Primär- und Fremdschlüsseln

```
1  "Nested Loop  (cost=29.96..371.06 rows=10 width=38)"
2  "  -> Hash Join  (cost=29.96..235.16 rows=370 width=46)"
3  "        Hash Cond: (p.userid = u.userid)"
4  "        -> Seq Scan on participation p  (cost
       =0.00..164.00 rows=10000 width=16)"
5  "        -> Hash  (cost=29.50..29.50 rows=37 width=38)"
6  "              -> Seq Scan on "user" u  (cost=0.00..29.50
       rows=37 width=38)"
7  "                    Filter: ((birthday >= '1986-01-01'::
       date) AND (birthday <= '1992-12-31'::date) AND (gender =
        'w'::text))"
8  "  -> Index Scan using event_pkey on event e  (cost
       =0.00..0.35 rows=1 width=8)"
9  "        Index Cond: (eventid = p.eventid)"
10 "        Filter: ((date >= '2013-01-01'::date) AND (date <=
        '2013-03-01'::date) AND (((numbermaleconfirmed /
       numberfemaleconfirmed))::numeric < 0.5))"
```

An diesem Ausführungsplan ist erkennbar, dass der Planer die Entscheidung, welche Tabelle treibend und welche innen sein soll, korrekt entscheidet. Das nested loop so wie es hier geplant ist, verursacht vorraussichtlich Kosten von 370 Zeilen

* 0,38 = 129,5. Hätte der Planer treibende und innere Tabelle vertauscht, wären Kosten in Höhe von 1 * 235,16 = 235,16 entstanden. Kann der Planer auf einen Index Scan zugreifen der eine hohe Selektivität hat, dann entscheidet er sich für einen nested loop join. Greift er mit einem Sequential Scan die Daten ab, dann hasht er sie.

sowie Indexen auf Fremdschlüssel- und Nicht-Id-Spalten:

Listing 2.29: Ausführungsplan Query 2 mit Indexen

```
1  "Nested Loop  (cost=18.70..359.80 rows=10 width=38)"
2  "  -> Hash Join  (cost=18.70..223.90 rows=370 width=46)"
3  "        Hash Cond: (p.userid = u.userid)"
4  "        -> Seq Scan on participation p  (cost
       =0.00..164.00 rows=10000 width=16)"
5  "        -> Hash  (cost=18.24..18.24 rows=37 width=38)"
6  "              -> Bitmap Heap Scan on "user" u  (cost
       =4.98..18.24 rows=37 width=38)"
7  "                    Recheck Cond: ((birthday >=
       '1986-01-01'::date) AND (birthday <= '1992-12-31'::date)
       )"
8  "                    Filter: (gender = 'w'::text)"
9  "                    -> Bitmap Index Scan on user_birthday
         (cost=0.00..4.97 rows=72 width=0)"
10 "                          Index Cond: ((birthday >=
       '1986-01-01'::date) AND (birthday <= '1992-12-31'::date)
       )"
11 "  -> Index Scan using event_pkey on event e  (cost
       =0.00..0.35 rows=1 width=8)"
12 "        Index Cond: (eventid = p.eventid)"
13 "        Filter: ((date >= '2013-01-01'::date) AND (date <=
        '2013-03-01'::date) AND (((numbermaleconfirmed /
       numberfemaleconfirmed))::numeric < 0.5))"
```

In diesem Ausführungsplan kommt erstmals ein Bitmap Index Scan zum Einsatz. Das ist möglich, da auf user.birthday indiziert wurde. Da alle Nutzer gesucht werden, deren Geburtstag im Bereich 1986-01-01 und 1992-12-31 liegen, bietet sich ein Bitmap Index Scan an.

Query 3: Die Anzahl der textmessages gruppiert und absteigend sortiert nach numberFemaleConfirmed, die zwischen dem 01.01.2010 und dem 31.12.2012 geschrieben wurden, in denen das Wort Salsa vorkommt, deren Empfänger männlich sind und zwischen 1972 und 1982 geboren wurden, deren Absender weiblich sind und die zwischen 1986 und 1992 geboren worden und an einem event teilnehmen, dass zwischen dem 01.01.2013 und dem 01.03.2013 in einem 100km Radius zu der GPS-Koordinate 11.5833 45.15.00 stattfindet und bei dem numberMaleConfirmed / numberFemaleConfirmed kleiner 0,5 ist. Das Ergebnis soll nur die ersten fünf Treffer liefern.

Listing 2.30: Query 3

```
1  Select e.numberFemaleConfirmed, Count(m.messageid) as
       anzahlMessages
2  from public.user u, public.event e, public.participation p,
        public.message m
3  where u.gender = 'w'
4  AND u.birthday between '01.01.1986' AND '31.12.1992'
5  AND e.date between '01.01.2013' AND '01.03.2013'
6  AND m.date between '01.01.2010' AND '31.12.2012'
7  AND m.textmessage like '%Salsa'
8  AND e.eventid = p.eventid
9  AND p.userid = u.userid
10 AND m.senderId = u.userId
11 AND e.numberMaleConfirmed / e.numberFemaleConfirmed < 0.5
12 AND DEGREES(acos(cos(RADIANS(90-e.lat))*cos(RADIANS(90-70))
       +sin(RADIANS(90-e.lat))*
13 sin(RADIANS(90-70))*cos(RADIANS(e.lon-80))))/360*40074 <
       10000
14 AND m.recipientId in
15 (Select u2.userId
16 from public.user u2
17 where u2.gender = 'm'
18 AND u2.birthday between '01.01.1972' AND '31.12.1982'
19 )
20 Group by e.numberFemaleConfirmed
```

```
21 Order by e.numberFemaleConfirmed DESC
22 Limit 5
```

Listing 2.31: Ausführungsplan für Query 3

```
1  "Limit  (cost=1386.84..1386.86 rows=1 width=12)"
2  "  -> GroupAggregate  (cost=1386.84..1386.86 rows=1 width
       =12)"
3  "        -> Sort  (cost=1386.84..1386.84 rows=1 width=12)"
4  "              Sort Key: e.numberfemaleconfirmed"
5  "              -> Nested Loop  (cost=269.91..1386.83 rows
       =1 width=12)"
6  "                    Join Filter: (m.senderid = p.userid)"
7  "                    -> Hash Join  (cost=239.79..1069.16
       rows=3 width=20)"
8  "                          Hash Cond: (e.eventid = p.
       eventid)"
9  "                          -> Seq Scan on event e  (cost
       =0.00..829.00 rows=92 width=12)"
10 "                                Filter: ((date >=
       '2013-01-01'::date) AND (date <= '2013-03-01'::date) AND
        (((numbermaleconfirmed / numberfemaleconfirmed))::
       numeric < 0.5) AND (((degrees(acos(((cos(radians((90::
       double precision - lat))) * 0.939692620785908::double
       precision) + ((sin(radians((90::double precision - lat))
       ) * 0.342020143325669::double precision) * cos(radians((
       lon - 80::double precision))))))) / 360::double
       precision) * 40074::double precision) < 10000::double
       precision))"
11 "                          -> Hash  (cost=235.16..235.16
       rows=370 width=24)"
12 "                                -> Hash Join  (cost
       =29.96..235.16 rows=370 width=24)"
13 "                                      Hash Cond: (p.userid
        = u.userid)"
14 "                                      -> Seq Scan on
       participation p  (cost=0.00..164.00 rows=10000 width=16)
```

```
    "
15  "                                          -> Hash  (cost
    =29.50..29.50 rows=37 width=8)"
16  "                                               -> Seq Scan
    on "user" u  (cost=0.00..29.50 rows=37 width=8)"
17  "                                                     Filter:
    ((birthday >= '1986-01-01'::date) AND (birthday <=
    '1992-12-31'::date) AND (gender = 'w'::text))"
18  "                    -> Materialize  (cost=30.13..314.39
    rows=77 width=16)"
19  "                          -> Hash Semi Join  (cost
    =30.13..314.01 rows=77 width=16)"
20  "                                Hash Cond: (m.recipientid
    = u2.userid)"
21  "                                -> Seq Scan on message m
     (cost=0.00..279.00 rows=1533 width=24)"
22  "                                      Filter: ((date >=
    '2010-01-01'::date) AND (date <= '2012-12-31'::date) AND
     (textmessage ~~ '%Salsa'::text))"
23  "                                -> Hash  (cost
    =29.50..29.50 rows=50 width=8)"
24  "                                     -> Seq Scan on "
    user" u2  (cost=0.00..29.50 rows=50 width=8)"
25  "                                           Filter: ((
    birthday >= '1972-01-01'::date) AND (birthday <=
    '1982-12-31'::date) AND (gender = 'm'::text))"
```

Listing 2.32: Ausführungsplan Query 3 mit Primär- und Fremdschlüsseln

```
1  mit Primär- und Fremdschlüsseln für alle Tabellen:
2  "Limit  (cost=546.93..546.95 rows=1 width=12)"
3  "  -> GroupAggregate  (cost=546.93..546.95 rows=1 width
   =12)"
4  "        -> Sort  (cost=546.93..546.93 rows=1 width=12)"
5  "              Sort Key: e.numberfemaleconfirmed"
6  "              -> Nested Loop  (cost=333.57..546.92 rows=1
    width=12)"
```

```
7  "                        ->  Hash Join  (cost=333.57..535.37
   rows=28 width=16)"
8  "                              Hash Cond: (p.userid = u.userid)
   "
9  "                              ->  Seq Scan on participation p
    (cost=0.00..164.00 rows=10000 width=16)"
10 "                              ->  Hash  (cost=333.53..333.53
   rows=3 width=24)"
11 "                                    ->  Nested Loop Semi Join
    (cost=29.96..333.53 rows=3 width=24)"
12 "                                          ->  Hash Join  (cost
   =29.96..315.28 rows=57 width=32)"
13 "                                                Hash Cond: (m.
   senderid = u.userid)"
14 "                                                ->  Seq Scan
   on message m  (cost=0.00..279.00 rows=1533 width=24)"
15 "                                                      Filter:
   ((date >= '2010-01-01'::date) AND (date <=
   '2012-12-31'::date) AND (textmessage ~~ '%Salsa'::text))
   "
16 "                                                ->  Hash  (
   cost=29.50..29.50 rows=37 width=8)"
17 "                                                      ->  Seq
   Scan on "user" u  (cost=0.00..29.50 rows=37 width=8)"
18 "
   Filter: ((birthday >= '1986-01-01'::date) AND (birthday
   <= '1992-12-31'::date) AND (gender = 'w'::text))"
19 "                                          ->  Index Scan using
    user_pkey on "user" u2  (cost=0.00..0.32 rows=1 width
   =8)"
20 "                                                Index Cond: (
   userid = m.recipientid)"
21 "                                                Filter: ((
   birthday >= '1972-01-01'::date) AND (birthday <=
   '1982-12-31'::date) AND (gender = 'm'::text))"
```

```
22 "                        -> Index Scan using event_pkey on
    event e  (cost=0.00..0.40 rows=1 width=12)"
23 "                             Index Cond: (eventid = p.eventid
    )"
24 "                             Filter: ((date >= '2013-01-01'::
    date) AND (date <= '2013-03-01'::date) AND (((
    numbermaleconfirmed / numberfemaleconfirmed))::numeric <
     0.5) AND (((degrees(acos(((cos(radians((90::double
    precision - lat))) * 0.939692620785908::double precision
    ) + ((sin(radians((90::double precision - lat))) *
    0.342020143325669::double precision) * cos(radians((lon
    - 80::double precision)))))))) / 360::double precision) *
     40074::double precision) < 10000::double precision))"
```

sowie Indexen auf Fremdschlüssel- und Nicht-Id-Spalten:

Listing 2.33: Ausführungsplan Query 3 mit Indexen

```
1  "Limit  (cost=355.27..355.29 rows=1 width=12)"
2  "  -> GroupAggregate  (cost=355.27..355.29 rows=1 width
    =12)"
3  "        -> Sort  (cost=355.27..355.28 rows=1 width=12)"
4  "              Sort Key: e.numberfemaleconfirmed"
5  "              -> Nested Loop  (cost=18.70..355.26 rows=1
    width=12)"
6  "                    -> Nested Loop  (cost=18.70..343.72
    rows=28 width=16)"
7  "                          -> Nested Loop Semi Join  (cost
    =18.70..322.27 rows=3 width=24)"
8  "                                -> Hash Join  (cost
    =18.70..304.02 rows=57 width=32)"
9  "                                      Hash Cond: (m.
    senderid = u.userid)"
10 "                                      -> Seq Scan on
    message m  (cost=0.00..279.00 rows=1533 width=24)"
11 "                                            Filter: ((date
     >= '2010-01-01'::date) AND (date <= '2012-12-31'::date)
```

```
      AND (textmessage ~~ '%Salsa'::text))"
12 "                                    -> Hash (cost
    =18.24..18.24 rows=37 width=8)"
13 "                                          -> Bitmap
    Heap Scan on "user" u (cost=4.98..18.24 rows=37 width
    =8)"
14 "                                                Recheck
    Cond: ((birthday >= '1986-01-01'::date) AND (birthday <=
     '1992-12-31'::date))"
15 "                                                Filter:
    (gender = 'w'::text)"
16 "                                                ->
    Bitmap Index Scan on user_birthday (cost=0.00..4.97
    rows=72 width=0)"
17 "
    Index Cond: ((birthday >= '1986-01-01'::date) AND (
    birthday <= '1992-12-31'::date))"
18 "                                -> Index Scan using
    user_pkey on "user" u2 (cost=0.00..0.32 rows=1 width=8)
    "
19 "                                    Index Cond: (userid
    = m.recipientid)"
20 "                                    Filter: ((birthday
    >= '1972-01-01'::date) AND (birthday <= '1982-12-31'::
    date) AND (gender = 'm'::text))"
21 "                            -> Index Scan using
    participation_userid on participation p (cost
    =0.00..7.02 rows=10 width=16)"
22 "                                Index Cond: (userid = u.
    userid)"
23 "                        -> Index Scan using event_pkey on
    event e (cost=0.00..0.40 rows=1 width=12)"
24 "                            Index Cond: (eventid = p.eventid
    )"
25 "                            Filter: ((date >= '2013-01-01'::
    date) AND (date <= '2013-03-01'::date) AND (((
```

```
numbermaleconfirmed / numberfemaleconfirmed))::numeric <
  0.5) AND (((degrees(acos(((cos(radians((90::double
precision - lat))) * 0.939692620785908::double precision
) + ((sin(radians((90::double precision - lat))) *
0.342020143325669::double precision) * cos(radians((lon
- 80::double precision)))))))) / 360::double precision) *
  40074::double precision) < 10000::double precision))"
```

2.8 Was tun bei langsamen Ausführungsplänen?

- Um dem Planer keine falschen Hinweise zu geben, sollten keine Hints verwendet werden.
- Autoanalyze aktivieren
  Der Planner kann nur dann optimierte Ausführungspläne erzeugen, wenn er genügend Statistiken über die gefüllten Tabellen besitzt. Deswegen müssen regelmäßig Statistiken erstellt werden. In PostgreSQL geht das mit analyze. Analyze sammelt Informationen über den Füllstand der Tabellen, die häufigsten Werte in jeder Spalten und die wahrscheinliche Verteilung der Werte in einer Spalte. Mit diesen Statistiken kann der Planer dann den passenden JOIN-Algorithmus und die passende JOIN-Order wählen. Um nach bestimmten Abständen automatisch analyze aufzurufen, gibt es autoanalyze.
- Autovacuum aktivieren
  Mit autovacuum = on in der postgresql.conf kann autovacuum aktiviert werden.
- Indexe verwenden
  Es kann z.B. der Entwickler geeignete Indexe anlegen, sodass das DBMS diese verwenden kann, um performantere Ausführungspläne zu erzeugen.
- Mehrspaltige Indexe verwenden, wenn dadurch ein performanterer Ausführungsplan erzeugt werden kann.
- Keine zu intensiven Rechnungen in SQL formulieren.
- Partitioning verwenden.

- Die Struktur der Tabellen überdenken, wenn mehr als 8 Tabellen miteinander verknüpft werden.
- Herausfinden, warum der Planner einen langsamen Plan erzeugt, anstatt durch Planner-Hints der Frage aus dem Weg zu gehen.
- Es kann sein, dass jemand ungünstige Werte für die Parameter in der postgresql.conf gesetzt hat.
- Bei einem zu kleinen Wert für *join_collapse_limit* in der postgresql.conf, verbunden mit der expliziten Verknüpfung von Tabellen mit dem Wort Join, ist der Planner gezwungen, eine vorgegebene aber u.U. ungünstige Join-Reihenfolge zu verwenden.
- Um dem Planner nicht ausversehen zu etwas zu zwingen, sollten bei einem Innerjoin die Tabellen nicht explizit mit dem Wort JOIN verknüpfen werden, sondern es sollten die Tabellen einfach getrennt durch ein Komma angeben werden:

Listing 2.34: Inner-Join

```
1 SELECT * FROM a, b, c WHERE a.id = b.id AND b.ref = c.
    id;
```

- Bei einem zu kleinen Wert für *from_collapse_limit* in der postgresql.conf, verbunden mit vielen Subqueries, kann der Planner die Subqueries nicht auflösen. Dabei sollte der Planner Subqueries zu Joins auflösen, da sonst erst das komplette Ergebnis des Subquerys erstellt werden muss, bevor mit der Tabelle weitergearbeitet werden kann.

2.9 Zusammenfassung

Der Planer von PostgreSQL ist so ausgereift, dass dessen Entwickler bewusst auf ein Hints-System verzichten. Es wird davon ausgegangen, dass der Planer umso bessere Ausführungspläne macht, je mehr Entscheidungsfreiheit er bei der Planerstellung hat. Diese Entscheidungsfreiheit würden durch Hints eingeschränkt. Ausführungsplanoptimierung beschränkt sich bei PostgreSQL vor allem darauf, sicherzustellen, dass der Planer immer aktuelle Statistiken hat und ihm möglichst

viele Indexe zur Verfügung stehen, aus denen er die seiner meiner nach Besten wählen kann, um einen effizienten Ausführungsplan zu erstellen.

2.10 Ausblick

In einem weiteren Projekt könnte man die Ausführungspläne bei Tabellen mit Partitioning untersuchen.

Es könnte auch die Arbeitsweise des Planers untersucht werden, wenn mehr als fünfzehn Tabellen miteinander verknüpft werden und er mit einer genetischen Suche nach einem effizienten Aufführungsplan sucht.

Es könnte gezeigt werden, welchen Einfluss der Füllstand einer Tabelle und die Verteilung der Werte in einer einzelnen Spalte auf die Generierung eines Ausführungsplanes hat.

2.11 Die Abarbeitung von Abfragen in PostgreSQL

1. **Empfang des SQL-Befehls**
   Nachdem der SQL-Befehl über eine Netzwerkverbindung übertragen wurde, findet die Kodierungsumwandlung statt und die weiteren Phasen der Abarbeitung sehen den Befehl in der Servercodierung. Hierbei gibt es nur sehr geringe Optimierungsmöglichkeiten. Es können theoretisch CPU-Zyklen gespart werden, wenn die Clientkodierung gleich der Servercodierung ist, ansonsten wird eine Konvertierung durchgeführt. Diese Auswirkungen sind jedoch sehr gering. Der Parameter *client_encoding* informiert den Server darüber, welche Kodierung die ankommenden Befehle haben und welche Kodierung das Anfrageergebnis haben soll, welches an den Client gesendet wird. Die Voreinstellung gibt an welche Kodierung der Server intern verwendet.

2. **Parser**
   In dieser Abarbeitungsphase wird die kodierte Befehlszeichenkette durch einen internen Parse-Baum dargestellt. Des weiten wird die Befehlszeichenkette auf semantische Bedingungen überprüft und etwas bearbeitet. Die

SQL-Befehle werden dann aufgeteilt in sogenannte optimierbare Anweisungen(SELECT, INSERT, UPDATE und DELETE) und Hilfsanweisungen. Die Hilfsanweisungen werden später direkt ausgeführt und sie erzeugen keine Ausgabe. Dagegen kommen die optimierbaren Anweisungen in den Rewriter. Für den Parser gibt es von der Anwenderseite keine Möglichkeit die Geschwindigkeit zu optimieren.

3. **Query Rewriter**
   Der Rewriter wendet die Anfrageumschreibregeln(Query Rewrite Rules) an. Dabei werden die Sichten(Views) und andere benutzerdefinierte Regeln aufgelöst, in die Anfrage eingebaut und im Parse-Baum ersetzt. Da der Rewriter vor dem Planer angesiedelt ist, bekommt der Planer es nicht mit, ob die Anfrage aus einer Sicht kam oder nicht. Mit der Erstellung einer Sicht hat man somit keinen Optimierungsvorteil.

4. **Planer / Optimizer**
   Der Planer bekommt den möglicherweise umgeschriebenen Parse-Baum und hat die Aufgabe einen Ausführungsplan(execution plan) zu erstellen, der ebenfalls ein Baum ist. Der Ausführungsplan beschreibt wie auf die Tabellen zugegriffen werden soll, also welche Indexe und Join-Algorithmen verwendet werden sollen und in welcher Reihenfolge. Es soll möglichst der optimalste und schnellste Ausführungsplan gefunden werden.

5. **Executor**
   Der vom Planer auserwählte Ausführungsplan wird vom Executor ausgeführt. Dabei werden Zugriffsrechte auf Tabellen und andere Objekte und Constraints geprüft. Die Laufzeit der Ausführung hängt nicht nur davon ab ob der Plan gut ist, sondern auch von der gesamten Systemkonfiguration.

A Codebeispiele

Listing A.1: alle Tabellen erstellen

```
1  CREATE TABLE "user"
2  (
3    userid bigint,
4    name text,
5    email text,
6    gender text,
7    birthday date,
8    password text,
9    image text
10 )
11 WITH (
12   OIDS=FALSE
13 );
14 ALTER TABLE "user"
15   OWNER TO postgres;
16
17 CREATE TABLE event
18 (
19   eventid bigint NOT NULL,
20   creatorid bigint,
21   date date,
22   eventname text,
23   occasion text,
24   location text,
25   lon double precision,
26   lat double precision,
27   description text,
```

```
28    numbermaleconfirmed int,
29    numberfemaleconfirmed int
30 )
31 WITH (
32    OIDS=FALSE
33 );
34 ALTER TABLE event
35    OWNER TO postgres;
36
37 CREATE TABLE message
38 (
39    messageid bigint,
40    eventid bigint,
41    senderid bigint,
42    recipientid bigint,
43    textmessage text,
44    date date
45 )
46 WITH (
47    OIDS=FALSE
48 );
49 ALTER TABLE message
50    OWNER TO postgres;
51
52 CREATE TABLE participation
53 (
54    participationid bigint,
55    userid bigint,
56    eventid bigint
57 )
58 WITH (
59    OIDS=FALSE
60 );
61 ALTER TABLE participation
62    OWNER TO postgres;
```

Listing A.2: Datenimport über COPY

```
1 COPY public.Event (eventid, creatorid, date, eventname,
    occasion, location, lon, lat, description,
    numbermaleconfirmed, numberfemaleconfirmed) From 'C:\
    Event.txt' DELIMITER ';';
2 COPY public.Message (messageid, eventid, senderid,
    recipientid, textmessage, date) From 'C:\Message.txt'
    DELIMITER ';';
3 COPY public.Participation (participationid, userid, eventid
    ) From 'C:\Participation.txt' DELIMITER ';';
4 COPY public.User (userId, name, email, gender, birthday,
    password, image) From 'C:\User.txt' DELIMITER ';';
```

Listing A.3: Primär- und Fremdschlüssel hinzufügen

```
1 ALTER TABLE public.event ADD PRIMARY KEY (eventid);
2 ALTER TABLE public.message ADD PRIMARY KEY (messageid);
3 ALTER TABLE public.participation ADD PRIMARY KEY (
    participationid);
4 ALTER TABLE public.user ADD PRIMARY KEY (userid);
5
6 ALTER TABLE event ADD CONSTRAINT event_creatorid FOREIGN
    KEY (creatorid) REFERENCES public.user (userid) MATCH
    FULL;
7 ALTER TABLE message ADD CONSTRAINT message_eventid FOREIGN
    KEY (eventid) REFERENCES event (eventid) MATCH FULL;
8 ALTER TABLE message ADD CONSTRAINT message_senderid FOREIGN
     KEY (senderid) REFERENCES public.user (userid) MATCH
    FULL;
9 ALTER TABLE message ADD CONSTRAINT message_recipientid
    FOREIGN KEY (recipientid) REFERENCES public.user (userid
    ) MATCH FULL;
10 ALTER TABLE participation ADD CONSTRAINT
    participation_userid FOREIGN KEY (userid) REFERENCES
    public.user (userid) MATCH FULL;
11 ALTER TABLE participation ADD CONSTRAINT
    participation_eventid FOREIGN KEY (eventid) REFERENCES
```

```
event (eventid) MATCH FULL;
```

Listing A.4: Indexe auf Spalten legen

```
1  CREATE INDEX event_creatorid ON public.event(creatorid);
2  CREATE INDEX message_eventid ON public.message(eventid);
3  CREATE INDEX message_senderid ON public.message(senderid);
4  CREATE INDEX message_recipientid ON public.message(
       recipientid);
5  CREATE INDEX participation_userid ON public.participation(
       userid);
6  CREATE INDEX participation_eventid ON public.participation(
       eventid);
7
8  CREATE INDEX event_date ON public.event(date);
9  CREATE INDEX event_eventname ON public.event(eventname);
10 CREATE INDEX event_occasion ON public.event(occasion);
11 CREATE INDEX event_location ON public.event(location);
12 CREATE INDEX event_lon ON public.event(lon);
13 CREATE INDEX event_lat ON public.event(lat);
14 CREATE INDEX event_numbermaleconfirmed ON public.event(
       numbermaleconfirmed);
15 CREATE INDEX event_numberfemaleconfirmed ON public.event(
       numberfemaleconfirmed);
16
17 CREATE INDEX message_textmessage ON public.message(
       textmessage);
18 CREATE INDEX message_date ON public.message(date);
19
20 CREATE INDEX user_name ON public.user(name);
21 CREATE INDEX user_email ON public.user(email);
22 CREATE INDEX user_gender ON public.user(gender);
23 CREATE INDEX user_birthday ON public.user(birthday);
```

B Arbeitsaufteilung

Arbeit	C. Ochmann	I. Körner
Abstract		0

Tabelle B.1: Aufteilung vom Abstract

Arbeit	C. Ochmann	I. Körner
Einleitung		1.1
Aufgabenstellung		
Forschungsgegenstand		
akt. Wissensstand		
Eingesetzte Datenbank		
Projektplanung		
Anwendungsfälle		
EasyMock		
Dependency Injection		

Tabelle B.2: Aufteilung von Kapitel 2

Arbeit	C. Ochmann	I. Körner
Datengenerator		

Tabelle B.3: Aufteilung von Kapitel 3

Arbeit	**C. Ochmann**	**I. Körner**
Ausblick		

Tabelle B.4: Aufteilung von Kapitel 6

C Eigenständigkeitserklärung

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe. Mir ist bekannt, dass jede Form des Plagiats mit der Note 5 (Betrugsversuch) bewertet wird.

Ochmann, Christof Unterschrift:

Körner, Ingo Unterschrift: