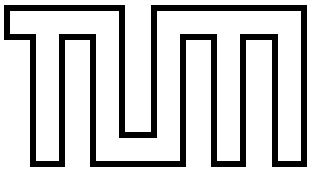


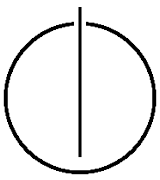
FAKULTÄT FÜR INFORMATIK

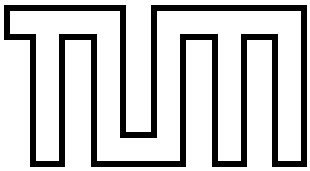
DER TECHNISCHEN UNIVERSITÄT MÜNCHEN

Bachelorarbeit in Informatik

Ein Bash-zu-SQL-Compiler für die in-situ Dateianalyse

Maximilian E. Schüle





FAKULTÄT FÜR INFORMATIK

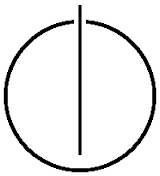
DER TECHNISCHEN UNIVERSITÄT MÜNCHEN

Bachelorarbeit in Informatik

Ein Bash-zu-SQL-Compiler für die in-situ Dateianalyse

A Bash to SQL Compiler for in-place File Analysis

Autor:	Maximilian E. Schüle
Themensteller:	Professor Alfons Kemper, Ph.D.
Betreuer:	Tobias Mühlbauer
Datum:	15. Januar 2015



Ich versichere, dass ich diese Abschlussarbeit selbständig verfasst und nur die angegebenen
Quellen und Hilfsmittel verwendet habe.

München, den 12. November 2014 Maximilian E. Schüle

Acknowledgments

If someone contributed to the thesis... might be good to thank them here.

Abstract

Performance tests on query processing on flat file databases using bash and on modern relational databases using SQL and developing a Bash to SQL compiler to substitute command line tools

Zusammenfassung

Performanzvergleich der Anfrageverarbeitung auf textbasierten Datenbanken mittels Skriptsprachen (Bash mit awk und sed) zu modernen Datenbanksystemn mittels SQL und Entwicklung eines Bash-zu-SQL-Übersetzers zur Ersetzung von Zeilenkommandos

Inhaltsverzeichnis

Outline of the Thesis

Part I: Introduction and Theory

CHAPTER 1: INTRODUCTION

This chapter presents an overview of the thesis and it purpose. Furthermore, it will discuss the sense of life in a very general approach.

CHAPTER 2: THEORY

No thesis without theory.

Part II: The Real Work

CHAPTER 3: OVERVIEW

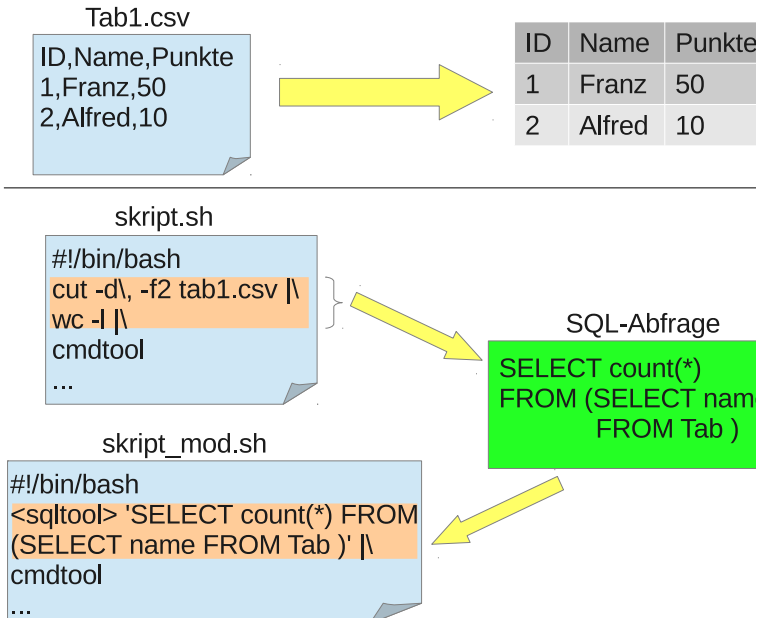
This chapter presents the requirements for the process.

Teil I.

Motivation

1. Einführung

Here starts the thesis with an introduction. Please use nice latex and bibtex entries [7]. Do not spend time on formating your thesis, but on its content.



1.1. Latex Introduction

There is no need for a latex introduction since there is plenty of literature out there.

Teil II.

Bash statt SQL

2. Bash statt SQL

Das Ziel ist zwar, am Ende die Bash-Kommandos zu ersetzen, aber um die Kommandos zu ersetzen, wird zuerst die Idee benötigt, wie sähe denn eigentlich so eine SQL-Anfrage in der Bash aus, also nur mithilfe der klassischen Unix-Befehle, die textbasierte Datenbanken auslesen. Daher erklärt dieses Kapitel zuerst, wie solche Datenbanken nur von Kommandos wie cat, cut, awk und sed analog zu SQL-Abfragen ausgelesen werden können. Anschließend werden mit diesen Befehlen Datenbank-Benchmarks implementiert und die Zeit der Abfragen bei großen Datenmengen gemessen. Der Vergleich mit neueren Programmen erfolgt dann im nächsten Kapitel.

2.1. Textbasierte Datenbanken

Das Jahr 1970 bedeutete einen Umbruch im Bereich der Datenbanken, Edgar F. Codd veröffentlichte sein Papier über das relationale Modell, der Grundlage aller relationaler Datenbanken. Es erlaubt, mehrere Relationen zu verbinden, ohne dass der Benutzer sich um die interne Repräsentation kümmern muss.[3] Dennoch dauerte es weitere neun Jahre bis Larry Ellison und Bob Miner den ersten Prototyp einer relationalen Datenbank mit ihrer Firma Software Development Laboratories auf den Markt brachten, bei IBM sogar zwei Jahre länger.[6] Da stellt sich die Frage, wie wurden Datensätze, die die Grundlage der Datenverarbeitung darstellen, zuvor abgespeichert - die Antwort: textbasierte Datenbanken (engl.: flat file databases).

Franz Winkler	Am Winkl 5, 80000 Musterstadt
Xaver Ziegler	Maurergasse 19, 80000 Musterstadt

Abbildung 2.1.: Beispiel für textbasierte Datenbank

Textbasierte Datenbanken gibt es schon immer, sobald eine Person ein Adressbuch von Kontakten mit Namen und Adressen pflegt, so ist das eine textbasierte Datenbank. Sobald ein Kontakt den Wohnsitz wechselt oder die Anzahl an Freunden wächst, muss die Datenbank aktualisert werden, ist sie geordnet, bedeutet das in manchen Fällen, die Daten komplett neuzuschreiben, ein unnötiger Aufwand.
Mit diesem Problem konfrontiert hat der deutschstämmige US-Amerikaner Hermann Hollerith bereits 1884 das erste Datenformat erfunden,[8, S.48] abgeschaut von Schaffnern in Zügen, die die Fahrkarten an bestimmten Stellen lochten, um Wiederverwendung auszuschließen, entwickelte er die Lochkarte. Das System wurde 1890 erstmals bei einer Volkszählungen eingesetzt, wobei die Lochkarten die Informationen über Geschlecht und Alter enthielten und elektrische Maschinen in 3,4 Sekunden eine Karte auslasen. Weitere Aufträge auf der ganzen Erde folgten, Hollerith gründete die Computing Tabulating Recording

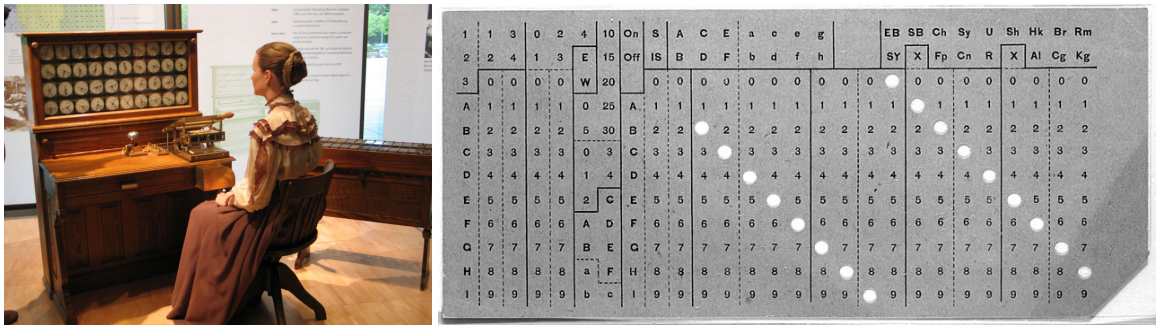


Abbildung 2.2.: Hollerith Maschine [2] mit Lochkarte [wikipedia.de]

```
root:x:0:0:root:/root:/bin/bash
daemon:x:1:1:daemon:/usr/sbin:/bin/sh
bin:x:2:2:bin:/bin:/bin/sh
sys:x:3:3:sys:/dev:/bin/sh
sync:x:4:65534:sync:/bin:/bin/sync
```

Abbildung 2.3.: Auszug aus /etc/passwd eines Unix-Systems mit Trennzeichen ':'

Company (CTR), aus der 1924 schließlich IBM hervorging.[2]

Textbasierte Datenbanken sind mittlerweile weit verbreitet, in Unix-Systemen (entwickelt Mitte der 1960-er Jahre) werden so Daten zum Beispiel zur Passwort- oder Gruppenverwaltung gespeichert. (siehe Abb. 2.3) Jeder Datensatz wird durch einen Zeilenvorschub (engl.: newline) abgeschlossen, einzelne Felder durch ein Trennzeichen (engl.: delimiter) separiert.[4]

Ein Beispiel für textbasierte Datenbanken sind CSV-Dateien (Comma-Separated Values) manchmal auch DSV-Dateien (Delimiter-Separated-Values), also durch Komma oder anderes Trennzeichen separierte Tabellen, bei denen in den meisten Fällen ein Komma oder ein Semikolon die Spalten trennt. Erstmals erwähnt als Teil der Fortran Spezifikation zwischen 1968 und 1972 für listenorientierte Ein- und Ausgabe (Common Format and MIME Type for Comma-Separated Values (CSV) Files) [1, S. 17], haben sich CSV-Dateien mittlerweile zum Standard im Datenaustausch entwickelt. Inzwischen existiert sogar eine Richtlinie für CSV-Dateien, herausgegeben von der Internet Engineering Task Force in der RFC4180. [10] Nach dieser Richtlinie sollen CSV-Dateien folgende Regeln einhalten:

- jeder Datensatz ist in einer Zeile gespeichert, beendet durch Zeilenvorschub
- der letzte Datensatz benötigt keinen Zeilenvorschub
- die erste Zeile kann der Bezeichner sein
- die Felder sind durch Kommata getrennt, nach der letzten Spalte folgt kein Komma
- jedes Feld kann in Anführungszeichen stehen

- Kommata und Zeilenvorschübe als Teil eine Feldes müssen in Anführungszeichen eingeschlossen sein
- sind Anführungszeichen Teil des Formats, müssen Hochkommata (als Inhalt von Feldern) durch eine Fluchtsequenz gekennzeichnet sein

Die Vorteile solcher Datenbanken liegen in ihrer Einfachheit, sie speichern alle Informationen, es werden keine weiteren Informationen benötigt, es ist leicht, damit Informationen zu importieren und exportieren, da außer einem Trennzeichen und einem finalen Zeilenvorschub keine Konventionen einzuhalten sind.

Die Nutzung textbasierte Datenbanken bei Abfragen birgt aber Nachteile, so lassen sich die Daten speichern und verschicken, ein Ändern einzelner Datensätze erweist sich als schwierig, ohne die komplette Datei zu überschreiben. Da das Format nur wenigen Beschränkungen unterliegt, enthält in manchen Fällen die erste Zeile die Feldbezeichner, in anderen bereits den ersten Inhalt. Aber das gravierendere Problem liegt in der Anzahl der Datensätze, jeder Datensatz muss zeilenweise ausgelesen werden, da auch keine Konventionen befolgt werden, ist auch nicht von einer Sortierung auszugehen. Somit ist die Performanz solcher Datenbanken auch um einiges schlechter, wie dieses Kapitel anhand von Benchmarks noch zeigt.

Punktetabelle:

ID	Name	Punkte
1	Franz	50
2	Alfred	10
3	Marie	27

Zeittabelle:

ID	Zeit
1	44
2	88
3	67

Abbildung 2.4.: Beispiel Datenbank

2.2. Relationale Algebra der Unix-Shell im Vergleich zu SQL

Welche Strukturen in eines Shell-Skripts sind in welche SQL-Anweisungen zu übersetzen? Um dies besser vergleichen zu können, werden im Folgenden Ausdrücke der relationalen Algebra als Befehle in der der Unix-Shell ausgedrückt und eine äquivalente Abfrage in SQL angegeben. Als Grundlage für das relationale Algebra dienen Operatoren aus [14] und in diesem Kapitel werden ausschließlich die grundlegenden Kommandos der Unix-Shell verwendet, heute auch bekannt als GNU core utilities.[5]

2.2.1. Grundlage

Zuerst einmal liegt eine solche textbasierte Datenbank vor, bevor relationale Algebra angewandt wird, soll nur der Inhalt ausgegeben werden, nutzt ein einfacher SQL-Befehl wie:

```
SELECT * FROM Punktetabelle
```

Solche Anfragen können mit Befehlen wie *cat*, *more*, *less*, ... übersetzt werden und finden sich häufig, wenn vorher Daten durchgepiped werden.

```
cat Punktetabelle;
```

2.2.2. Selektion

Wenn jetzt Tupel ausgewählt werden, die ein Prädikat erfüllen sollen, also entspricht dies der Selektion, dann sind zwei Fälle zu unterscheiden: Äquivalenz: $\sigma_{ID=3}(Tabelle)$ und Vergleich $\sigma_{ID<3}(Tabelle)$ oder in SQL:

```
SELECT * FROM Punktetabelle WHERE ID=3;
SELECT * FROM Punktetabelle WHERE ID<3
SELECT * FROM Punktetabelle WHERE Name='Marie'
```

Die allgemeine Lösung nutzt *awk*, mit dem alle Vergleichsfunktionen einer höheren Programmiersprache implementiert sind, hierbei sind die Felder durch *$1,...,$n* bezeichnet, *$0* bezeichnet alle Felder, die Option *-F,* bezeichnet das Feldtrennzeichen (Delimiter), anschließend folgt ein Muster und der Befehl des ausgeführt wird (*' pattern {CMD}'*), in diesem Fall zuerst die Bedingung *1==3* und der Befehl der Ausgabe (*print $0*), in diesem Fall alle Spalten (das *SELECT* * der SQL).

```
awk -F, '$1==3 { print $0 }' Punktetabelle.csv;
awk -F, '$1>3  { print $0 }' Punktetabelle.csv;
awk -F, '$2="Marie" { print $0 }' Punktetabelle.csv;
```

Andere grundlegende Kommandos funktionieren meist nur bei kompletter Äquivalenz, wie *grep*, das in einer Datei nach allen Vorkommen der gewünschten Zeichenfolge sucht. Bei *sed* ist die die Äquivalenz auch einfach einzugeben, dabei ist es aber von Vorteil, zumindest den vorderen und hinteren Spaltentrenner mit anzugeben, oder gar alle möglichen:

```
grep -r '3,.*' Punktetabelle.csv;
grep -r 'Marie' Punktetabelle.csv;
sed -nr '/3,.*/p' Punktetabelle.csv;
sed -nr '/Marie/p' Punktetabelle.csv;
```

2.2.3. Projektion

Wenn nun einzelne Spalten ausgewählt werden, so wird die Projektion benötigt:
$\Pi_{Name,Punkte}(Punktetabelle)$

oder in SQL:

```
SELECT Name FROM Punktetabelle
```

Das klassische Unix-Kommando dazu ist *cut*, das es mit Hilfe der Option $-f$ erlaubt, einzelne Felder zu extrahieren, Felder werden beginnend beim ersten durch Aufzählung mit Kommata bestimmt (1,3) und ganze Bereiche mit Bindestrich ausgewählt (1-3 entspricht Feldern eins bis drei), der Spaltentrenner wird durch die Option *-d* mitgeteilt (Standard: Leerzeichen).

```
cut -f2,3 -d, Punktetabelle.csv;
```

Die Kommandos *awk* und *sed* erlauben die Projektion auch, ersterer Befehl einfach mit *print*, bei sed müssen explizit die Spaltentrenner angegeben werden:

```
awk -F, '{print $2,$3}' OFS=, Punktetabelle.csv;
sed -nr 's/([^,]*),([^,]*),(.*)/\2\3/p'
```

2.2.4. Vereinigung

Die Vereinigung $\Pi_{ID}(Punktetabelle) \cup \Pi_{ID}(Zeittabelle)$ ist am einfachsten in der Unix-Shell zu realisieren, schließlich unterstützt fast jeder Befehl durch Eingabe mehrerer Dateien das Zusammenfügen dieser. Für das Zusammenfügen oder Konkatenieren drängt sich *cat* (concatenate) geradezu auf, dadurch definiert sich doch dieser, einfach alle Dateien der Reihe nach auflisten:

```
cat datei1 datei2;
```

Und schon sind sie vereinigt, analog das Beispiel der oben gezeigten Projektion:

```
SELECT ID FROM Punktetabelle
UNION
SELECT ID FROM Zeittabelle
```

Das Beispiel erfordert vorher die Selektion, daher werden zwei anonyme Pipes verwendet, das sieht dann so aus:

```
cat <(cut -f2 -d, Punktetabelle.csv) \
    <(cut -f2 -d, Zeittabelle.csv);
```

2.2.5. Kreuzprodukt

Will man in der Shell das Kreuzprodukt $Punktetabelle \times Punktetabelle$ bilden, so geschieht das in SQL durch Auswahl mehrerer Tabellen:

```
SELECT * FROM Punktetabelle, Punktetabelle
```

In der Shell hilft einem auch hier *awk* weiter, diesmal mit Feldern. Zuerst werden alle Eingabezeilen (leeres Suchmusster) oder nur die gewünschten wie bisher durchgegangen und in dem Feld aufsteigend gespeichert. Anschließend, also im Schlussteil (bezeichnet durch END), kann mit den Feldern alles produziert werden, das Kreuzprodukt erfolgt durch die Ausgabe mit print in einer doppelten For-Schleife.

```
cat Punktetabelle | awk -F\| '
        {
                lines[i++]=$0
        }
        END{
                for (i in lines)
                  for (j in lines)
                    print lines[i], lines[j]
        }
' OFS=,
```

2.2.6. Mengendifferenz

Um alle Mengenoperationen der Algebra abzudecken, wird auch noch die Differenz benötigt, geschrieben als $R - S$, zum Beispiel ergibt $\Pi_{ID}(Punktetabelle) - \Pi_{ID}(Zeittabelle)$ die jenigen Tupel, zu denen kein passender Eintrag in der Zeittabelle einthalten ist.

```
SELECT ID FROM Punktetabelle
EXCEPT
SELECT ID FROM Zeittabelle
```

In der Unix-Shell gibt der Befehl *comm* die Zeilen in drei Spalten aus, zuerst die nur der ersten Datei (1), dann die nur der Zweiten (2) und dann die aus beiden (3), durch Angabe der Zahlen, können die Spalten unterdrückt werden. Für den Vergleich müssen die Dateien aber sortiert sein. Analog zur Algebra entspricht folgender Befehl der Differenz:

```
comm -23 R S;
```

Angewandt auf die Beispieltabellen:

```
comm -23 <(cut -f1 -d, Punktetabelle.csv | sort) \
         <(cut -f1 -d, Zeittabelle.csv | sort);
```

2.2.7. Umbennenung

Um alle Ausdrücke der relationalen Algebra abzudecken, fehlt nun noch die Umbenennung der Tabelle $\rho_{t1}(Punktetabelle)$ und einzelner Spalten $\rho_{Nr \leftarrow ID}(Punktetabelle)$. Da in der Shell die Tabellen nichts anders als Datenströme sind, ist keine Unterscheidung der Namen notwendig, eine Möglichkeit, die Tabellen umzubenennen, besteht nur darin, eine tempäre Hilfstabelle mit neuem Namen anzulegen.

```
cp Punktetabelle.csv t1.csv
```

Auch einzelne Spalten können nur mit ihrer Nummer angesprochen werden ($1,$2, etc.), eine Umbenennung kann nur für die Ausgabe erfolgen, der Strom muss also vorher mit *awk* oder *sed* bearbeitet werden.

```
sed -r 's/ID/Nr/';
cat Punktetabelle | awk -F\| '
        NR==1{
                print "Nr", $2, $3
        }
        NR>2{
                print $0
        }
' OFS=,
```

2.2.8. Relationale Verbund

Alle grundlegenden Operatoren der relationalen Algebra können auch mit einfachen Skripts auf textbasierten Datenbanken erfolgen, dennoch darf ein wichtiger Operator nicht fehlen,

der relationale Verbund (Join), vor allem der natürliche Verbund (natural join), der Tabellen über Äquivalenz zusammengehöriger Attribute verknüpft:

$R \bowtie S$
oder im Beispielfall mit Verknüpfung über ID: $Punktetabelle \bowtie Zeittabelle$ Der analoge Fall in SQL:

```
SELECT *
FROM Punktetabelle, Zeittabelle
WHERE Punktetabelle.ID = Zeittabelle.ID
```

In Unix stehen für Äqui-Joins jeglicher Art, bei denen jeweils ein Feld jeder Tabelle übereinstimmen sollen, das Kommando *join* zur Verfügung. Sollen zwei CSV-Dateien miteinander verknüpft werden, so müssen das Spaltentrennzeichen und die zu verknüpfenden Spalten (*-1 spalteA -2 spalteB* angegeben werden, standardmäßig der Leerraum (Whitespace) sowie die jeweils erste Spalte, und, sofern die die Zeile die Bezeichner enthält, müssen diese als solche mie *–header* deklariert sein.

```
join --header -t, -1 1 -2 1 Punktetabelle.csv Zeittabelle.csv
```

Zu beachten ist, dass im Ergebnis eine der verbundenen Spalten dann fehlt, also oben stehende Abfrage produziert folgendes Ergebnis:

ID	Name	Punkte	Zeit
1	Franz	50	44
2	Alfred	10	88
3	Marie	27	67

Die auszugebenden Spalten können auch hinter der Option *-o* explizit angegeben werden, *0* ist die verbundene Spalte, alle anderen mit der Spaltenummer der jeweiligen Tabelle, *2.3* meint die dritte Spalte der zweiten Tabelle. Also sollen die Spalten, für die die Join-Bedingung gilt, angezeigt werden und die zweite und dritte, so hilft folgender Befehl:

```
join -t, -o 0 1.2 1.3 tabelleA tabelleB
```

Damit können auch Semi-Joins $R \ltimes S$ und $R \rtimes S$ produziert werden, indem die Spalten angegeben sind, für einen linken Semi-Join sind das *-o 1.1 1.2 . . . 1.n* und für den rechten *-o 2.1 2.2 . . .*

Ein Join der Shell ist ein Sort-Join, er funktioniert (wie *comm* auch) nur auf sortierten Dateien, folglich muss oft eine Sortierung mit *sort* erfolgen, bevor gejoint werden kann. Das Kommando *sort* arbeitet mit Quicksort,[9] also mit Durchschnittslaufzeit $O(n\ log\ n)$, in schlechten Fällen auch $O(n^2)$. Dabei muss der Sortierfunktion noch das Trennzeichen (*-t,*) sowie die zu sortierenden Spalten als Feld angegeben werden, also *-k2,3* sortiert nach dem zweiten und dritten Feld. Erfolgt ein Join danach, so ist zu empfehlen, nach exakt einer Spalte zu sortieren *-k2,2*, da das Ergebnis sonst von der Länge des nachfolgenden Textes abhängt.

```
sort -t, -k2,2 tabelleA | join -1 2 - tabelleB
```

sort sortiert aber auch die Kopfzeile mit, also muss diese separat behandelt werden, am besten wird die erste Zeile mit *head* extrahiert, alle anderen mit *tail* und nach dem Sortieren können die Zeilen wieder zusammengefügt werden.

```
head -1 tabelleA > nurKopf
tail -n+2 tabelleA | sort -t, -k1,1 | cat nurKopf - > tabAmod
head -1 tabelleB > nurKopf2
tail -n+2 tabelleB | sort -t, -k1,1 | cat nurKopf2 - |\
join -t, -1 1 -2 1 tabAmod -
```

Auch der Antijoin $R \triangleright S$ bzw. $R \triangleright S$ ist in der Unix-Shell mit der Option *-v1* und *-v2* implementiert, der die Zeilen der ersten Tabelle (bzw. zweiten) ausgibt, zu denen kein Partner in der anderen Tabelle gefunden wurde. Damit nur die benötigte Spalten ausgegeben werden, empfiehlt es sich, die Spalten mit der Option *-o* noch explizit anzugeben:

```
join -t, -1 1 -2 1 -v1 -o 0 1.2 1.3 R.csv S.csv
```

Wenn zum Beispiel die Name ausgegeben werden sollen, zu denen keine Zeit gemessen wurde, sieht das so aus:

```
join -t, -1 1 -2 1 -v1 -o 1.2 Punktetabelle.csv Zeittabelle.csv
```

Als einzige Join-Arten, die noch fehlen, verbleiben die äußeren Joins $R \boxdot\!\bowtie\!\boxdot S$, $R \boxdot\!\bowtie S$ und $R \bowtie\!\boxdot S$, die auch der Unix-Befehl mit der Option *-a1* und *-a2* erzeugt, wodurch alle Zeilen der ersten (analog der zweiten) Datei ausgegeben werden, auch solche mit fehlendem Join-Partner. Die fehlenden Werte, bei SQL die NullWerte, können mit *-e "Wert"* angegeben werden, sollen sie mit "0" aufgefüllt werden, dann mit *-e "0"*.

```
join -t, -a1 -a2 -1 2 -2 2 -o 0 1.1 2.1 -e "0" tabelleR tabelleS
join -t, -a1     -1 2 -2 2 -o 0 1.1 -e "0" tabelleR tabelleS
join -t,     -a2 -1 2 -2 2 -o 0 2.1 -e "0" tabelleR tabelleS
```

2.2.9. Gruppierung und Aggregation

Über die relationale Algebra hinaus, geht der Gamma-Operator, der die Werte gruppiert und Aggragatsfunktionen wie max, min, sum oder avg auf ihnen erlaubt. So gibt
$\gamma_{count(*)}(Punktetabelle)$
die Anzahl aller Teilnehmer aus. Aus dem Standard-Reportoire der Unix-Shell ist auch der Befehl *awk* nützlich: Dazu sollten die Dateien vorher nach den Feldern sortiert sein, der Trick nutzt die Sortierung aus, die Werte der Spalten, nach denen gruppiert wird, wird vermerkt, die Aggregierung beginnt. Sobald sich ein Wert verändert, ist also zur nächsten Gruppe gesprungen worden, die aggregierten werden ausgegeben, die nächste Gruppe folgt, bis schließlich keine Zeilen mehr nachkommen, im END-Teil werden die letzten Aggregate ausgegeben.

```
SELECT spalte2, spalte3,
       max(spalte4), min(spalte4), count(*), avg(spalte4)
FROM Tabelle
GROUP BY spalte2, spalte3
```

Die Abfrage in *awk* übersetzt benutzt temporäre Variablen für die Summe, die Anzahl, das Minimun und das Maximum, der Durchschnitt setzt sich später aus Summe und Anzahl zusammen. Zudem werden die Werte gespeichert, nach denen gruppiert wird. Ändern sich diese nicht, so werden die Werte der aktuellen Zeile in der Aggragation ergänzt, *count* wird inkrementiert, der entsprechende Wert zur Summe von *sum* addiert und nach größer und kleiner für *min* und *max* geschaut. Passen die Werte zum Gruppieren nicht überein, so werden die alten ausgegeben und die Aggregationsvariablen zurückgesetzt.

```
head -1 tmp.csv > tmp1.csv
tail -n+2 tmp.csv  | sort -t\| -k2,2 | cat tmp1.csv - | awk -F\| '
        NR==1{print $2, $3,
                "max(S4)", "min(S4)", "count(*)", "avg(S4)"
        }
        NR==2{g2=$2; g3=$3; count=1; max4=$4; min4=$4; sum4=$4}
        NR>2{
                if( g2==$2 && g3==$3 ){
                        count++; sum4+=$4;
                        if(max4<$4)
                                max4=$4;
                        if(min4>$4)
                                min4=$4;
                }else{
                        print g2,g3,
                                max4,min4,sum4,count,sum4/count;
                        g2=$2; g3=$3;
                        count=1; max4=$4; min4=$4; sum4=$4
                }
        }
        END{print g2,g3,max4,min4,sum4,count,sum4/count}
' OFS=\|
```

Eine einfachere Lösung bietet der Befehl *uniq -c*, sofern nur die Anzahl der Vorkommnisse gezählt werden sollen.

2.3. Performanzmessungen

Das vorherige Kapitel hat die Grundlagen erklärt, also wie die relationale Algebra, auf der die relationale Anfragesprache SQL basiert, auf textbasierte Datenbanken angewandt werden kann. Dieses Kapitel behandelt die Performanz solcher Abfragen, also wie schnell sie sich ausführen lassen, auch im Vergleich zu modernen relationalen Datenbanken.

2.3.1. TPC-H Benchmarks

Um die Leistungsfähigkeit von Datenbanken zu testen wurde im Jahr 1988 auf Initiative von Omri Serlin hin ein Konsortium namens Transaction Processing Performance Council (TPC) gegründet, an dem acht Firmen der IT-Branche beteiligt waren.[11] Das Ziel war es nicht, "die Funktionen und Operationen von Rechnern zu testen, [sondern] Transaktionen

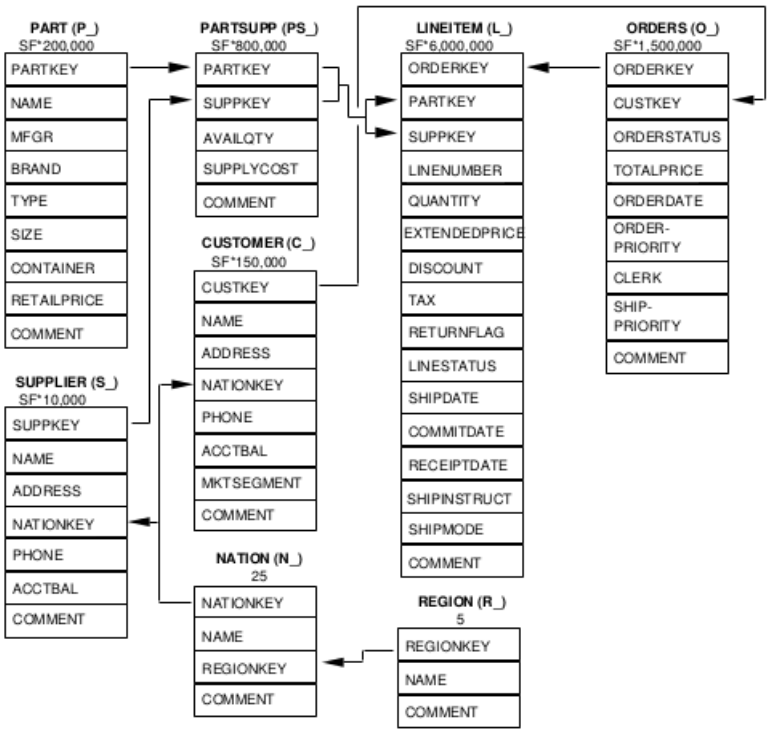


Abbildung 2.5.: TPC-H Schema [13]

zu betrachten, wie sie allgemein in der Geschätswelt üblich sind: Der Tausch von Gütern, Dienstleistungen und Geld."[12] So wurde der erste Benchmark für Datenbanksystem entwickelt, genannt TPC-A, der die maximalen Transaktionen pro Sekunde misst, wenn von verschiedenen Endgeräten darauf zugegriffen wird. Der Anwendungsbereich der TPC-A Benchmark ist die Online-Verarbeitung von Transaktionen, *Online Transaction Processing* (OLTP), wie sie in potentiellen Handelsunternehmen vorkommen, die Güter und Dienstleistungen gegen Geld tauschen. Sie "[zeichnen sich aus] durch relativ kurze Transaktionen, die im Allgemeinen nur auf ein eng begrenztes Datenvolumen zugreifen."[14, S. 711]

Der aktuellste Standard für OLTP-Anwendungen ist die TPC-H Benchmark, der die Leistung der Datenbank bei üblichen Anfragen misst, ohne dass die Datenbank zuvor darauf vorbereitet wird. Dazu sind 22 verschiedene Anfragen gegeben und eine Datenbasis, die mittels eines gegebenen Zufallsgenerators generiert wird, aber sich immer nach dem Handelsunternehmensschema aus acht Relationen richtet (vgl. Abb. 2.5).

Der Generator *DBGen* erzeugt die Datenbasis in verschiedenen Größen mit unterschiedlich vielen Tupeln in Abhängigkeit eines Faktors *SF* der ungefähr der Größe aller Daten in GB entspricht. So sind die möglichen Größen 1GB, 10GB, 30GB, 100GB, 300GB, 1000GB, 3000GB, 10000GB, 30000GB und 100000GB an Daten die per Zufall erstellt werden.

Table Name	Cardinality	Length (in bytes)	Typical Table
SUPPLIER	10,000	159	2
PART	200,000	155	30
PARTSUPP	800,000	144	110
CUSTOMER	150,000	179	26
ORDERS	1,500,000	104	149
LINEITEM	6,001,215	112	641
NATION	25	128	<1
REGION	5	124	<1
Total	8,661,245		956

Abbildung 2.6.: Größe der Relationen bei Faktor SF=1 [13]

2.3.2. Implementierung mit Shell-Skripten

Um nun die Leistungsfähigkeit der Shell-Skripte auf textbasierten Datenbanken zu testen, braucht es drei Werkzeuge: die Daten, die Skripte und natürlich Referenzwerten - die Skripte werden von Hand erzeugt, die Grundlage für die Datenbasis ist dieselbe wie für den TPC-H-Benchmark der Hyper-Schnittstelle[1], so lassen sich die Ergebnisse auch gleich vergleichen.

Die Implementierung der SQL-Anfragen orientiert sich an dem vorgestellten Schema im letzten Unterkapitel, nachfolgend sei nur die vierte TPC-H-Anfrage vorgestellt, die Implementierungen der anderen erfolgt analog und sind im Anhang einzusehen. Die vierte Abfrage der Benchmark bewirkt Folgendes:

> Mit Hilfe dieser Anfrage soll überprüft werden, wie gut das Auftragsprioritätensystem funktioniert. Zusätzlich liefert sie eine Einschätzung über die Zufriedenstellung der Kunden. Dazu zählt die Anfrage die Aufträge im dritten Quartal 1993, bei denen wenigstens eine Auftragsposition nach dem zugesagten Liefertermin zugestellt wurde. Die Ausgabeliste soll die Anzahl dieser Aufträge je Priorität sortiert in aufsteigender Reihenfolge enthalten.[14, S. 717]

In SQL ausgedrückt sieht das so aus:

```
select
        o_orderpriority,
        count(*) as order_count
from
        orders
where
        o_orderdate >= date '1993-07-01'
        and o_orderdate < date '1993-10-01'
        and exists (
```

[1] http://hyper-db.de/interface.html

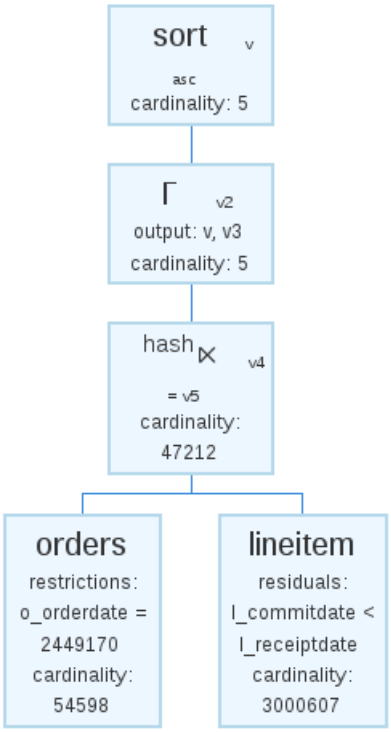


Abbildung 2.7.: Abfrageplan zu Nummer 4

```
                select
                        *
                from
                        lineitem
                where
                        l_orderkey = o_orderkey
                        and l_commitdate < l_receiptdate
        )
group by
        o_orderpriority
order by
        o_orderpriority
```

Um die Anfragen in Skripte zu übersetzen, hilft die Orientierung am Abfrage-Plan (siehe Abb. **??**). So erhält man einerseits den Ausdruck der relationalen Algebra dafür und eine Schritt-für-Schritt-Übersetzung ist möglich. Außerdem sind so die Ergebnisse besser vergleichbar. Die Tabellen liegen als CSV-Dateien mit Trennzeichen " | " in <tabellenname>.tbl vor, die Kopfzeilen analog in <tabellenname>.csv.

In diesem Fall müssen zuerst die Tabelle *orders* und *lineitem* nach den entsprechenden Tupeln gefiltert werden (o_orderdate >= date '1993-07-01' and o_orderdate < date '1993-10-01' und l_commitdate < l_receiptdate), in der Shell geschieht das am Schönste durch Auswahl der Zeilen mittels *awk*.

```
sort -k1,1 -t\| orders.tbl | cat orders.csv - | awk -F\| '
        NR==1{
                print $1"|" $6
        }
        NR>1 && $5<"1993-10-01" && $5>="1993-07-01"{
                print $1"|" $6
        }
' > tmporder.csv

sort -k1,1 -t\| lineitem.tbl |cat lineitem.csv - | awk -F\| '
        NR==1 || $12<$13{
                print $1
        }
' | uniq  |\
```

Das anschließende *exists* in SQL wird durch einen linken äußeren Join verwirklicht. Im Gegensatz zu einem Join soll nur auf Existenz überprüft werden, deshalb hilft das Unix-Kommando *uniq* aus, um Duplikate zu eliminieren.

```
join --header -t\| -1 1 -2 1 tmporder.csv - > tmp.csv
```

Im nächsten Schritt folgt die Gruppierung (*group by*) auf die Spalte *o_orderpriority*, die einfach durch *uniq* erfolgen kann, bei komplizierteren Aggragatsfunktionen (min, max, sum, count) ist es oft einfacher mit *awk* zu hantieren:

```
head -1 tmp.csv > tmp1.csv
tail -n+2 tmp.csv  | sort -t\| -k2,2 | cat tmp1.csv - | awk -F\| '
        NR==1{print $2,"order_count"}
        NR==2{g2=$2; count=1}
        NR>2{
                if( g2==$2 ){
                        count++
                }else{
                        print g2, count;
                        g2=$2;count=1;
                }
        }
        END{print g2,count}
' OFS=\|
```

Für den Fall der vierten TPC-H-Abfrage kann aber auch *uniq -c* benutzt werden, der neben der Gruppierung auch die Anzahl der Vorkommnisse aller Tupel mit ausgibt, dafür müssen wir uns zunächst auf die relevanten Spalten beschränken (mit *cut*) und alle Zeilen auch sortieren, damit der letztere Befehl auch ordentlich funktioniert:

```
cut -d\| -f2 | tail -n+2 | sort | uniq -c
```

Und schon ist das Skript fertig, beide Versionen liefern die gewünschte Ausgabe:

```
o_orderpriority | order_count
1-URGENT | 10594
2-HIGH | 10476
3-MEDIUM | 10410
4-NOT SPECIFIED | 10556
5-LOW | 10487
```

```
10594 1-URGENT
10476 2-HIGH
10410 3-MEDIUM
10556 4-NOT SPECIFIED
10487 5-LOW
```

Die Implementierung aller weitern TPC-H-Abfragen erfolgte analog.

query1	real 0m47.420s	user 0m39.370s	sys 0m2.744s
query10	real 0m12.971s	user 0m13.549s	sys 0m0.560s
query10p	real 0m12.374s	user 0m13.657s	sys 0m0.592s
query11	real 0m2.934s	user 0m3.300s	sys 0m0.112s
query11p	real 0m2.906s	user 0m3.284s	sys 0m0.096s
query12	real 0m22.859s	user 0m53.267s	sys 0m2.420s
query12p	real 0m21.855s	user 0m55.831s	sys 0m2.536s
query13	real 0m7.068s	user 0m8.089s	sys 0m0.256s
query13p	real 0m6.271s	user 0m8.601s	sys 0m0.252s
query14	real 0m7.185s	user 0m7.160s	sys 0m0.652s
query14p	real 0m6.767s	user 0m7.336s	sys 0m0.608s
query15p	real 0m29.505s	user 0m35.518s	sys 0m1.080s
query16	real 0m4.957s	user 0m5.692s	sys 0m0.432s
query16p	real 0m4.354s	user 0m6.104s	sys 0m0.496s
query17	real 0m37.156s	user 0m1.764s	sys 0m0.024s
query17p	real 0m37.317s	user 0m38.866s	sys 0m0.588s
query18	real 0m25.484s	user 0m34.946s	sys 0m0.972s
query18p	real 0m24.947s	user 0m35.770s	sys 0m0.892s
query19	real 0m6.728s	user 0m6.788s	sys 0m0.584s
query19p	real 0m6.443s	user 0m6.852s	sys 0m0.576s
query2	real 0m6.372s	user 0m6.540s	sys 0m0.320s
query20	real 0m58.405s	user 1m2.956s	sys 0m4.456s
query20p	real 0m59.412s	user 1m4.336s	sys 0m4.656s
query21	real 1m3.225s	user 1m11.084s	sys 0m1.308s
query21p	real 0m46.289s	user 1m16.401s	sys 0m1.228s
query22	real 0m6.178s	user 0m6.540s	sys 0m0.120s
query22p	real 0m5.921s	user 0m6.616s	sys 0m0.084s
query2p	real 0m5.974s	user 0m6.860s	sys 0m0.284s
query3	real 0m32.022s	user 0m43.503s	sys 0m3.428s
query3.alt	real 0m29.276s	user 0m41.879s	sys 0m3.320s
query3p	real 0m31.000s	user 0m43.975s	sys 0m3.352s
query4a	real 0m20.535s	user 0m54.887s	sys 0m2.548s
query4b	real 0m21.910s	user 0m53.483s	sys 0m2.380s
query5	real 0m26.827s	user 0m26.562s	sys 0m1.004s
query5p	real 0m24.696s	user 0m26.982s	sys 0m0.888s
query5pb	real 0m24.441s	user 0m26.634s	sys 0m0.796s
query6	real 0m7.949s	user 0m7.716s	sys 0m0.556s
query7	real 0m18.713s	user 0m30.002s	sys 0m1.580s
query7.alt	real 0m17.680s	user 0m24.814s	sys 0m0.940s
query7p	real 0m18.294s	user 0m30.270s	sys 0m1.540s
query8	real 0m39.677s	user 0m41.495s	sys 0m2.888s
query8p	real 0m37.294s	user 0m44.331s	sys 0m2.744s
query9	real 0m54.243s	user 1m9.584s	sys 0m2.580s
query9.alt	real 0m46.270s	user 0m49.831s	sys 0m1.848s
query9p	real 0m52.910s	user 1m10.184s	sys 0m2.480s

Abbildung 2.8.: Gemessene Zeiten der Abfragen mit SF=1 auf 2 Rechenkernen à 4 Threads

query1	real 9m32.857s	user 9m31.970s	sys 0m49.739s
query10	real 3m15.974s	user 3m24.642s	sys 0m8.759s
query10p	real 3m9.187s	user 3m26.610s	sys 0m9.261s
query11	real 0m34.194s	user 0m37.878s	sys 0m1.686s
query11p	real 0m34.220s	user 0m37.766s	sys 0m1.640s
query12	real 4m21.685s	user 8m54.478s	sys 0m26.273s
query12p	real 3m42.580s	user 9m1.861s	sys 0m27.183s
query13	real 1m40.206s	user 1m53.250s	sys 0m4.347s
query13p	real 1m30.957s	user 1m57.208s	sys 0m4.726s
query14	real 2m8.932s	user 2m9.678s	sys 0m12.308s
query14p	real 2m2.453s	user 2m10.616s	sys 0m12.022s
query15	real 7m26.731s	user 8m7.402s	sys 0m27.885s
query16	real 1m6.396s	user 1m11.125s	sys 0m6.569s
query16p	real 0m56.808s	user 1m13.359s	sys 0m7.201s
query17	real 7m48.630s	user 0m22.020s	sys 0m0.754s
query17p	real 7m52.637s	user 8m6.840s	sys 0m9.326s
query18	real 5m20.389s	user 6m47.595s	sys 0m20.564s
query18p	real 5m8.145s	user 6m57.919s	sys 0m20.536s
query19	real 2m55.907s	user 2m59.125s	sys 0m12.261s
query19p	real 2m52.960s	user 2m58.968s	sys 0m12.193s
query20	real 13m26.291s	user 13m18.047s	sys 1m30.816s
query20p	real 13m32.589s	user 13m27.336s	sys 1m31.657s
query21	real 14m51.314s	user 16m17.387s	sys 0m29.451s
query21p	real 10m42.894s	user 16m17.977s	sys 0m28.387s
query22	real 1m22.748s	user 1m27.505s	sys 0m2.001s
query22p	real 1m17.706s	user 1m27.889s	sys 0m1.971s
query2p	real 1m13.534s	user 1m22.012s	sys 0m4.298s
query3p	real 7m23.247s	user 9m18.016s	sys 0m59.532s
query4a	real 3m54.677s	user 9m26.891s	sys 0m28.586s
query4b	real 3m58.975s	user 8m46.921s	sys 0m27.085s
query5b	real 5m7.112s	user 5m26.370s	sys 0m17.300s
query5p	real 5m5.788s	user 5m24.228s	sys 0m17.527s
query6	real 2m20.614s	user 2m16.706s	sys 0m10.894s
query7	real 4m20.438s	user 5m55.684s	sys 0m33.536s
query7p	real 4m14.706s	user 5m57.255s	sys 0m33.023s
query8	real 8m38.944s	user 8m35.325s	sys 0m49.757s
query8p	real 8m12.842s	user 8m44.417s	sys 0m49.566s
query9	real 15m4.204s	user 17m10.921s	sys 0m52.472s
query9p	real 14m37.647s	user 17m4.015s	sys 0m51.840s

Abbildung 2.9.: Gemessene Zeiten der Abfragen mit SF=10

Teil III.

Vergleich existierender Tools für textbasierte Datenbanken

Teil IV.

Der Bash zu SQL Parser

Teil V.

Ausblick

Appendix

A. Detailed Descriptions

Here come the details that are not supposed to be in the regular text.

Literaturverzeichnis

[1] *IBM FORTRAN Program Products for OS and the CMS Component of VM/370 General Information*. IBM, 1972.

[2] Detlef Borchers. *Heinz Nixdorf Museumsforum zeigt Hollerithmaschine*. Heise Zeitschriften Verlag, 10.05.2007.

[3] Edgar F. Codd. A relational model of data for large shared data banks. 1970.

[4] Glenn Fowler. *cql - A Flat File Database Query Language*. AT&T Bell Laboratories, 1994.

[5] Jürgen Gulbins. *UNIX*. Springer-Verlag, 1985.

[6] Harry Henderson. *A to Z of computer scientists*. Facts On File, 2003.

[7] Leslie Lamport. *LaTeX : A Documentation Preparation System User's Guide and Reference Manual*. Addison-Wesley Professional, 1994.

[8] Jeremy O. Baum Marketa J. Zvelebil. *Understanding Bioinformatics*. Garland Science, 2008.

[9] Robert Sedgewick. *Computer Science 226: Data Structures and Algorithms*. Princeton University, 2002.

[10] Y. Shafranovich. *Common Format and MIME Type for Comma-Separated Values (CSV) Files*. Network Working Group, 2005.

[11] Kim Shanley. *History and Overview of the TPC*. TPC, 02.1998.

[12] TPC. *About the TPC*. TPC, 2014.

[13] Transaction Processing Performance Council (TPC). *TPC BENCHMARK H*. Transaction Processing Performance Council (TPC), 2013.

[14] Alfons Kemper und André Eickler. *Datenbanksysteme - Eine Einführung*. Oldenburg Verlag, 2011.