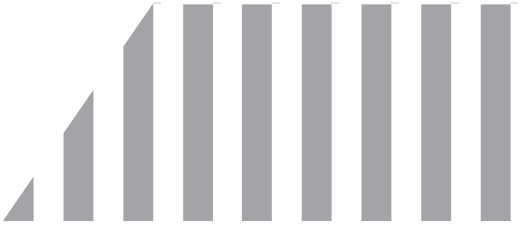
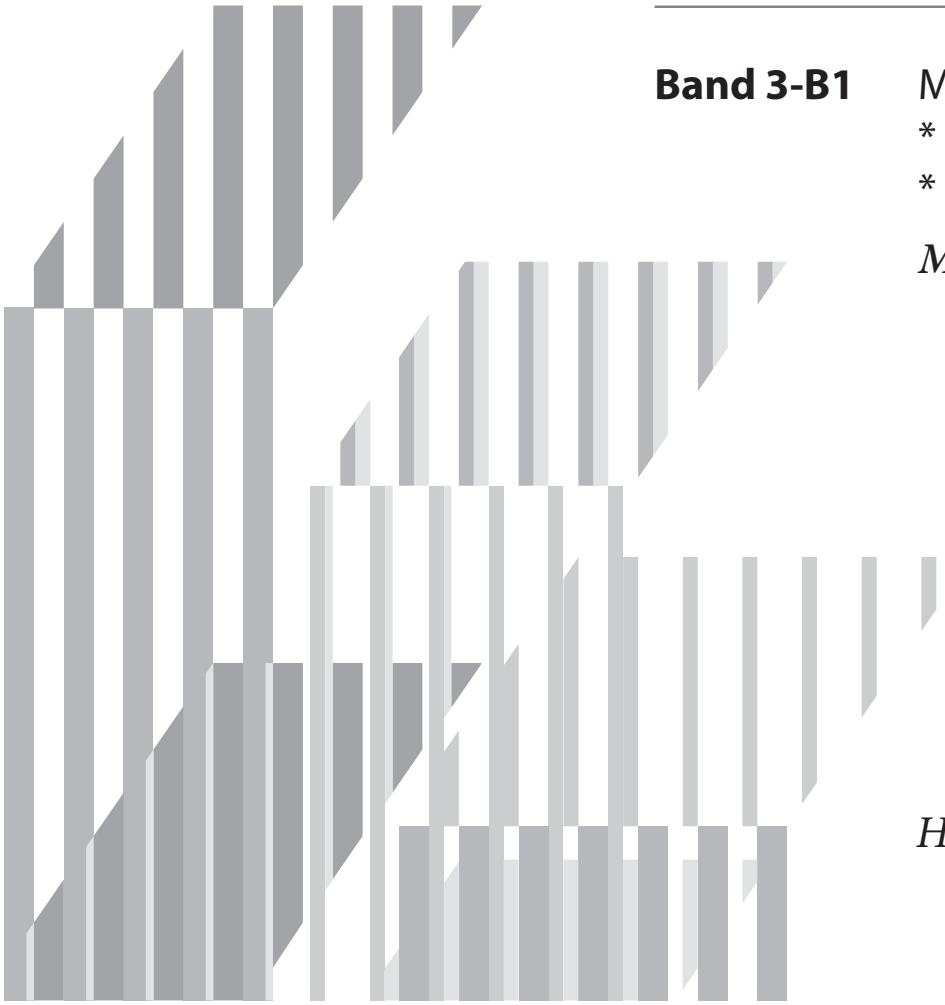
Band 3-B1



MODERNE STRUKTURIERTE PROGRAMMIERUNG

Objektorientiert und algorithmisch
ENTWERFEN | IMPLEMENTIEREN | TESTEN



Band 3-B1 Mischen
* sequentiell
* relational

Methode

Horst van Bremen

Bibliografische Information der Deutschen Nationalbibliothek:
Die Deutsche Nationalbibliothek verzeichnet diese Publikation in der Deutschen Nationalbibliografie; detaillierte bibliografische Daten sind im Internet über http://dnb.dnb.de abrufbar.

Gestaltung	Katharina Schmitt
Titel- und Einbandgrafik	Monika Sylvia Gomm † 1971
Englisch-Korrektorat	David Roseveare
Verlag	BoD · Books on Demand GmbH, Überseering 33, 22297 Hamburg, bod@bod.de
Druck	Libri Plureos GmbH, Friedensallee 273, 22763 Hamburg
Version / Datum	Version 1.0.2 vom 1.7.2026
Literaturverzeichnis	siehe Kapitel „11.1 Literaturverzeichnis“ auf Seite 297 f
Rechtliches	siehe Kapitel „11.2 Rechtliche Hinweise“ auf Seite 299 ff
ISBN des Ringbuchs	978-3-6967-2725-3

Über den Autor

Horst van Bremen
Diplom 1972 an der TU Darmstadt – Elektrotechnik und Informatik
39 Berufsjahre in der IT, davon 18 als Freiberufler

IT-Systemarchitekt
Datenbankexperte

Programmiererfahrung seit 1969
Strukturierte Programmierung nach Jackson seit 1974
Freier Autor seit 2011

Vorwort zur B1-Ausgabe

Im Band 1 wurde die Moderne Strukturierte Programmierung als eine Disziplin vorgestellt, die einem sich wiederholenden Schema folgt: Aus der Erkennung der an einem Verarbeitungsvorgang beteiligten Datenstrukturen und deren Prüfung auf Synchronisierbarkeit folgt die Ableitung von Modul- und Programmstrukturen, die nach der Bestückung mit den passenden Operationen als grafische Modelle des neu zu schreibenden Codes dienen. Dabei können durchaus Probleme auftauchen, die mit diesem Ansatz unlösbar sind, für die aber im folgenden Band 5 Lösungsalternativen angeboten werden. Diese liegen des öfteren nicht auf der Hand und erfordern daher eine gewisse Kreativität, von der bisher jedoch allenfalls bei der Link-Listen-Verarbeitung etwas spürbar war. Es ist folglich an der Zeit, das beschriebene Schema zu verlassen und die Modellierung als schöpferischen Akt zu betrachten, aus dem letztlich die Datenstrukturen folgen, soweit physisch möglich.

Dafür ist ein Aufgabenfeld erforderlich, das mehrere Lösungsvarianten zulässt, die zwar dasselbe Produkt zu liefern imstande sind, sich aber signifikant unterscheiden. Es kommt dabei also nicht primär auf das richtige Erkennen der beteiligten Datenstrukturen an, sondern auf die Verwendung der datenorientierten Modellierung als Kreativwerkzeug. Die erste Frage lautet also: Was ist möglich? – Nicht aber: Wie ist es beschaffen? Das kommt erst als zweiter Schritt, denn jede Variante muss natürlich realitätsfest sein.

Die Bände 3-B1 und 4-B1 demonstrieren diese – bislang ungewohnte – kreative Dimension der Jackson-Methode anhand des Mischens elementarer Datenbestände, die aus Key-Value-Kombinationen bestehen. Jeder komplexere Bestand, der an Mischvorgängen teilnimmt, lässt sich auf diese Grundform zurückführen, auch wenn eine Vielzahl an Regeln hinzutreten mag, die für jeden Einzelfall festlegen, wie damit zu verfahren ist. Bereits die in diesem Band zu Beginn vorgestellten – sehr kleinen und einfachen – Bestände ermöglichen es, darauf fünf sequentielle und sechs relationale Lösungsvarianten aufzubauen, deren Eigenschaften in diesem Band 3-B1 detailliert beschrieben werden.

Das ist kein Glasperlenspiel, sondern hat große praktische Bedeutung. In großen Anwendungssystemen kann nicht alles mit allem transaktional verzahnt sein. Vielmehr werden häufig Schnittstellen gebildet, in denen sich Änderungsbestände ansammeln, welche zyklisch oder anforderungsgetrieben in ihre Zielbestände eingearbeitet werden müssen. Das geschah in den Anfangsjahren der Datenverarbeitung rein sequentiell und wird heutzutage meistens datenbankorientiert realisiert, wobei die sequentiellen Verarbeitungsformen keineswegs völlig verschwunden sind, vor allem dann, wenn es sich um große Änderungsbestände handelt.

„Datenbankorientiert“ bedeutet üblicherweise: relational. Das ermöglicht elegante Lösungen, die früher nicht machbar gewesen wären. Allerdings: Die modernen Relational Database Management Systems (RDBMS) können zwar extrem große Datenbestände verwalten, aber sie haben eine Achillesferse, nämlich ihre relativ niedrigen Limite für gleichzeitig aktive Änderungsvorgänge. Neu eingefügte, geänderte oder gelöschte Zeilen müssen vom RDBMS gegen fremde Zugriffe so lange geschützt werden, bis die Änderungen freigegeben werden. Das ist bei Hunderten oder tausenden parallel stattfindenden ändernden Zugriffen um so aufwendiger, je größer die betreffenden Mengen sind. Jede Verarbeitung, die im Konkurrenzbetrieb viele Zeilen bearbeitet, muss diese deshalb in Teilmengen vertretbarer Größe aufteilen. Wenn sie zwischendrin abbricht oder abgebrochen wird, muss sie bei erneutem Start auf dem erreichten Stand aufsetzen, also wiederanlauffähig sein.

Die hier vorgelegten relationalen Varianten sind daher teilweise als restartfähig konzipiert. Da dieses Thema als besonders heikel gilt, wird das jeweilige Vorgehen genau erklärt und plausibel gemacht. Damit geht ein stilistischer Wandel einher, der das Spielerische des Bands 1 ablegt. Das macht die Lektüre zwar weniger amüsant, aber treibt die Professionalisierung voran. Mit dem Wissen aus diesem Band können Mischaufgaben mittleren Schwierigkeitsgrads identifiziert, Lösungen konzipiert und tragfähige Realisierungen vorbereitet werden. Dafür wünsche ich Ihnen, geschätzte Leserinnen und Leser, gutes Gelingen!

Lemgo, den 1.7.2026

Übersichtsverzeichnis Band 3-B1

Inhaltsverzeichnis Band 3-B1

Band 3

09/2015 Zusammenfluss zweier Seitenarme der Sorgue in L'Isle-sur-la-Sorgue (Okzitanien)

Mischen und Entflechten

Erst dadurch, dass Daten aus verschiedenen Quellen zusammenfließen, kann etwas völlig Neues entstehen. Treffen gleichartige Daten – in Analogie zum nebenstehenden Bild – aufeinander, dann geht es meistens um das Erzeugen zunehmend umfangreicher Datenbestände. Aus Rinnsalen werden Bäche, aus Bächen kleine Flüsse, aus denen große Flüsse entstehen, die schließlich in einem Meer münden. In der IT haben wir es analog hierzu mit „Data Flows" und „Data Lakes" zu tun. Sind dagegen verschiedenartige Objekte an Mischvorgängen beteiligt, dann wird es spannend.

Das Wort „Zusammenfließen" impliziert ein gleichsinniges Zusammenströmen, dem in späteren Kapiteln die Bezeichnung „synchrones Mischen" entspricht. Es gibt aber auch ganz andere Mischvorgänge in der IT, für die in der sichtbaren Realität keine unmittelbare Analogie existiert. Das Gegenteil des Mischens ist das Entflechten, das wir uns wiederum ganz einfach wie das Aufteilen eines Fließgewässers in mehrere Arme vorstellen können.

Auf dem ersten Blick mutet es seltsam an, dass einer solch überschaubaren Thematik eine ganze Staffel der Buchreihe gewidmet ist, aber im weiteren Verlauf wird es sich zeigen, dass die Dinge bei weitem nicht so einfach sind, wie sie anfangs zu sein scheinen.

1.1 Alter Wein in neuen Schläuchen?

Das Mischen und das Entflechten von Datenströmen gehören auch in modernen Anwendungssystemen zu den Basisprozessen. Sie werden zwar nicht immer so bezeichnet und sie haben keineswegs stets dieselbe wiedererkennbare Struktur wie beispielsweise die (Rang-)Gruppenverarbeitungen, aber das ist unwichtig. Es handelt sich hierbei um Oberbegriffe für eine Vielzahl von Vorgängen, bei denen Daten zusammengeführt oder voneinander getrennt werden. Dementsprechend wäre es verwegen, diese Fülle auch nur annähernd erfassen zu wollen. Das ist auch gar nicht nötig, denn es geht hier darum, die gemeinsamen Merkmale solcher Verarbeitungen erkennen und mit JSP-Mitteln beschreiben zu können.

Bevor damit begonnen werden kann, lohnt ein Blick in die „Umwelt", die mit den IT-Themen und -Begriffen oft herzlich wenig anzufangen weiß. Wikipedia, zum Beispiel, erklärt den Begriff „Mischen" so[1]:

Mischen oder **Mischungsverhältnis** steht für:

- Mischen (Spielkarten), Verfahren zur Randomisierung von Spielkarten
- Mischen (Verfahrenstechnik), eine der vier Hauptprozessgruppen der mechanischen Verfahrenstechnik
- Mischer (Elektronik), Erzeugen von Summen- und Differenzfrequenz in elektronischen Mischstufen
- Farbmischung, Zusammenführen von Farben oder Farbmitteln
- Vermengen von chemischen Reinstoffen zu einem Gemisch
  - Anteil des Wasserdampfes in der atmosphärischen Luft: Luftfeuchtigkeit#Feuchtegrad
- zufälliges Vermengen von Elementen oder Objekten, siehe Randomisierung
- das Versetzen von Mauerwerk, siehe Mauerwerksverband
- Verschlüsselung von Klartext durch zufällige Vertauschung, siehe Verschlüsselung
- Verfahren zum Zusammenstellen von Formationen und Spielpaarungen im Pétanque […]

1 Erneut abgerufen am 23.03.2026. Der Artikel wurde seit 2018 zwar editiert, aber inhaltlich hatte sich nur wenig verändert.

Unter „siehe auch" auf derselben Seite gibt es einen Verweis auf „Mischverfahren". Dieser lautet wie folgt:

> **Mischverfahren** steht für:
>
> Verfahren zum Mischen im Allgemeinen, siehe Mischen (Verfahrenstechnik)
> ein Ableitungssystem in der Entwässerungstechnik, siehe Mischsystem

Von IT keine Spur. Zu „Entflechtung" gibt es ebenfalls eine Wikipedia-Begriffsklärungsseite:

> Entflechtung bezeichnet:
> • im übertragenen Sinne einen Umstrukturierungsprozess bei Unternehmen […]
> • beim Layout (Entwurf) von Leiterplatten das kreuzungsfreie Verlegen von Leiterbahnen […]
> • bei der Eisenbahn ein Bauwerk zur konfliktfreien Verzweigung von Bahnlinien […]

Zudem gibt es zahlreiche Einzelartikel, in denen das Wort „entflechten" vorkommt, allerdings meistens ohne Bezug zur IT. Diese Beispiele belegen die Breite und damit die mangelnde Trennschärfe des Begriffs „Entflechtung". Also müssen wir diese Begriffe im IT-Kontext selbst klären.

Wenn wir heute noch einen Programmierer interviewen könnten, der in den fünfziger und sechziger Jahren des vergangenen Jahrhunderts seine beiden letzten Berufsjahrzehnte erlebte – wie würde der den Begriff „Mischen" definieren? Vielleicht so:

> »Anfangs haben wir Lochkarten mit einer elektromechanischen Sortiermaschine gemischt, bevor wir sie in den Kartenleser stapelten. Das jeweilige Programm hat die aufeinanderfolgenden und zusammengehörenden Karten in neue Daten umgesetzt, die dann mit dem Kartenstanzer ausgegeben wurden. Es gab ja damals noch keine Massenspeicher, sondern nur einige K Worte Hauptspeicher, wenn es hoch kam. Ausserdem kannte man noch keine Betriebssysteme im heutigen Sinn. Jedes Programm musste mit Interrupts umgehen können und dafür eigene Treiberfunktionen besitzen.
>
> Dann bekamen wir Bandgeräte und konnten anstatt mit Lochkarten mit Banddateien arbeiten. In denen mussten die – anfangs aus Karten stammenden – Sätze in ganz bestimmten Reihenfolgen stehen, die wir durch immer raffiniertere Sortierverfahren unter Beteiligung mehrerer Bandstationen erzeugten. Die Ausgabedateien wurden ebenfalls auf Band geschrieben. Dadurch konnten wir uns auch von der Begrenzung auf 80 Zeichen pro Lochkarte lösen, weil die Sätze auf den Bändern viel länger sein durften, aber nicht länger als die paarweisen Puffer in der Maschine, die wir für die Wechselpuffertechnik brauchten. Nur mit dieser Technik schafften wir es, die Bänder schnell genug zu beschreiben, ohne sie zwischendrin abstoppen zu müssen.
>
> Allerdings war es sehr ineffizient, die Daten satzweise zu lesen und zu schreiben, weil die Bandmaschinen zwischen je zwei Sätzen eine große Lücke als Bremsstrecke für die Mechanik ließen. Darum passten nicht so viele Daten auf ein einzelnes Band, wie wir es bei größeren Datenmengen gebraucht hätten, und wir mussten uns mit Folgebändern herumplagen.
>
> Erst als unsere Maschine gegen eine schnellere mit wesentlich mehr Speicher ausgetauscht wurde, konnten wir die Daten in Blöcke zusammenfassen, also viele – meistens gleich lange – logische Sätze zu einem einzigen physischen Satz zusammenpacken. Mit jedem I/O wurde ein ganzer Block gelesen oder geschrieben. Dadurch gab es viel weniger Lücken auf den Bändern und somit eine weit bessere Kapazitätsausnutzung. Die Bandstationen wurden auch schneller und die Bänder bekamen zugleich höhere Schreibdichten. Das Dumme war nur, dass immer mehr Daten anfielen und wir deshalb weiterhin Folgebänder brauchten. Zudem gab es ein Durcheinander wegen der unterschiedlichen Schreibdichten bei gemischter Nutzung alter und neuer Bänder.

Echte Probleme hatten wir mit sehr langen Sätzen, die das Blockformat nicht gut ausnutzten oder länger als ein Block waren. Wir haben diese dann in Stücke zerlegt, um möglichst wenige Blöcke zu bekommen, aber es war mühsam, die Stücke unmittelbar vor der Verarbeitung wieder zusammenzusetzen. Noch schlimmer waren variabel lange Sätze, die womöglich mehrere Blöcke überspannten…
Aber, Sie haben mich ja eigentlich nach dem Mischen gefragt. Da hat sich, als wir endlich die Bänder hatten, eigentlich nicht mehr viel dran geändert. Sie müssen sich das wie einen Reißverschluß vorstellen, bei dem die Zähne schön nacheinander im Verschlußteil zusammengeführt werden. Den Reißverschlußzähnen entsprechen die Sätze von zwei Banddateien. Wenn diese Sätze sauber sortiert sind, gibt es solche, deren Schlüssel paarig sind, und mit denen etwas Besonderes passiert. Dann werden beispielsweise zwei Zahlen addiert, die in diesen Sätzen enthalten sind, und es wird jeweils nur ein Satz mit der Summe ausgegeben. Die unpaarigen Sätze werden dagegen einfach an der richtigen Stelle in den Ausgabestrom eingereiht. Man kann das auch mit mehr als zwei Eingabedateien parallel machen, aber dann wird es rasch extrem unübersichtlich.«

Die schnelle Entwicklung der Sortier- und Mischverfahren wurde damals durch den Zwang angetrieben, mit – im Verhältnis zu heute – indiskutabel leistungsschwachen, aber sehr voluminösen Rechnern immer größere Datenmengen unter anderem für kommerzielle Zwecke zu verarbeiten. Ein Fachbuch von Hartmut Wedekind mit dem Titel „Datenorganisation“ [10], das 1970 erschien, widmet den Sortierverfahren 60 Seiten, bei insgesamt 271 Seiten also ein knappes Viertel! Da das Sortieren mit Hilfe externer Speicher – damals: im Allgemeinen Bandmaschinen – aus iterativen Mischvorgängen besteht, die unterschiedlich effektiv arbeiten und jeweils eine bestimmte maschinelle Ausstattung benötigen, wurden in diesem Fachbuch diverse Misch-Algorithmen vorgestellt. Damals stand auch schon deutlich mehr Hauptspeicher als in den Jahren davor zur Verfügung, was die Entwicklung von In Core Sorts antrieb.

Was geht uns das heute noch an? Jedes Smartphone hat mehr Speicher und eine schnellere CPU als ein damaliger Großrechner – ganz zu schweigen von den Maschinen, die derzeit an der Spitze der Leistungsskala stehen. Sortier- und Mischprozesse mit großen Datenmengen dürften bei solchen Boliden doch gar kein Problem mehr sein, das irgendwie Beachtung verdient. Das ist jedoch ein Irrtum.

In einem Projekt des Autors erhielten die Zeilen einer partitionierten Datenbanktabelle von ca. 0,5 TB Größe im Jahr 2007 einen Primärindex. Jahrelang war kein eindeutiger Index nötig gewesen; aufgrund einer neuen Anforderung an das betreffende Anwendungssystem wurde er aber unumgänglich. Die in 240 Partitionen gespeicherten Daten – jeweils ca. 2 GB – mussten dafür partitionsweise (unsortiert) entladen, sortiert, mit dem Primärschlüssel ergänzt und dann wieder partitionsweise geladen werden. Das dauerte auf einer damals modernen Anlage mit schnellen CPUs, reichlich Hauptspeicher und großem Plattenpool von Samstagmorgen bis Sonntagnachmittag! Da an Wochenenden wie diesem meistens mehrere dringende Langläufe eingeplant waren, hätte es kaum wesentlich länger dauern dürfen. Die heute diskutierten Größenordnungen im Petabyte-Bereich – 1 PB = 1.000 TB = 1.000.000 GB – sind mit solchen Großrechnern unerreichbar. 2.000 Wochenenden für 1 PB entsprächen ca. 38,5 Jahren…

Angesichts des anhaltenden Wachstums der Datenbestände vieler Großorganisationen – keineswegs nur Banken – ist die Effizienz beim Sortieren und Mischen nach wie vor ein zentrales Thema. Das gilt auch am unteren Ende der Mengenskala, nämlich für einzelne Transaktionen. Auch diese können als elementare Mischvorgänge aufgefasst werden, was Ihnen, geschätzte Leserinnen und Leser, vermutlich etwas seltsam vorkommen dürfte. Doch gemach! Zu den Details kommen wir noch. Höchste Transaktionsraten lassen sich jedenfalls nur in Systemen erreichen, die dafür sorgfältig vorbereitet wurden. Immer wieder wird in der Presse von Servern berichtet, die unter dem Ansturm von Interessenten zusammenbrechen, welche beispielsweise neu emittierte Aktien über die Börse kaufen wollen. Enorme finanzielle Schäden und Haftungsprobleme sind die Folge. Das trockene Thema „Mischen und Entflechten“ bekommt angesichts dieser Gefahren ein weitaus größeres Gewicht, als spontan zu erwarten wäre.

1.2 Mischen: Eine Positionsbestimmung

Unser fiktiver Programmierer aus der Kartenstapel- und Banddatei-Zeit hat eine feste Vorstellung vom Mischen: Man nehme zwei sortierte Dateien und verarbeite sie im Reißverschlussverfahren gegeneinander. Basta. Hmmm… Das kennen wir doch schon! EvaluateSportsDS verarbeitet die Sätze des School Dataset in Dreiergruppen jedesmal dann, wenn eine neue „r-school group" beginnt. Auch das ist offensichtlich eine Mischverarbeitung, allerdings eine spezielle, weil sie völlige Synchronität voraussetzt: Es darf nicht passieren, dass Daten zu einer Schule fehlen. Ebensowenig ist es zulässig, Schul-Daten-Dreiergruppen zu Schulen einzuschieben, die nicht am Wettbewerb teilnehmen. Nur unter dieser besonderen Bedingung konnten die Korrespondenzen – wie im Unterkapitel „Schul-Daten in einer zweiten Datei" des Band-1-A2-Kapitels „Ranggruppenverarbeitung I" gezeigt – gezeichnet und die Programmstruktur mit ihrer Hilfe korrekt abgeleitet werden.

Ganz genau genommen, müssen die „r-school groups" und die Dreiergruppen hinsichtlich der School ID nicht exakt aufsteigend oder absteigend sortiert sein. Es genügt, dass sie zueinander synchron wechseln und die neuen Schlüssel immer 1:1 zusammenpassen. Es ist aber wesentlich mühsamer, dann Synchronität zu garantieren, als die Sätze beider Dateien einfach in der School ID-Sequenz – aufsteigend – zu sortieren. Die Sätze im Sports Dataset müssen innerhalb derselben School ID sowieso nach „gender code" und „age" – aufsteigend – geordnet sein, weil sonst die Zählung der Schüler-Gruppen(größen) völlig durcheinanderkommt.

Vor Jahrzehnten hätten dieses Beispiel und die damit verwandten Konstellationen oder auch Komplikationen ausgereicht, um das Thema „Mischen" komplett abzuhandeln. Es wurde damals nahezu ausschließlich in einem sequentiellen Sinn verstanden. Zwar gab es schon in den siebziger Jahren des vergangenen Jahrhunderts Direktzugriffsdateien und erste – darauf aufbauende – Datenbanksysteme, aber die heutige, weit komplexere IT-Welt war noch nicht einmal in Umrissen erkennbar. In ihr wirken die Basisprozesse „Mischen" und „Entflechten" vielerorts und in Kontexten, die es nötig machen, eine neue Begriffsbestimmung vorzunehmen. Das geschieht im Folgenden nicht als akademischer Diskurs, sondern aus einer eher anwendungsorientierten Sicht, die immer wieder Bezug auf das Hauptthema dieses Buchs nimmt. Auf diesem Weg ist zwar keine flächendeckende Beschreibung dieser Basisprozesse möglich, aber deren Vielfalt trotz innerer Verwandtheit soll als Einstieg in eigene Überlegungen dienen, ob ein Ihnen vorliegendes Entwurfs- und Programmierproblem möglicherweise auch in diese Kategorie gehört und nach deren Regeln gelöst werden kann.

Die IT ist eine extrem junge Disziplin, verglichen mit den etablierten Wissenschaften, zeigt aber eine hohe Dynamik. Sie wird daher auch künftig zahlreiche Ideen, Methoden, Verfahren und Produkte hervorbringen, die wir uns heute noch gar nicht vorstellen können. Die Fülle der Mischprozesse wird dabei sicherlich ebenso beständig erweitert werden, und es gibt für sie schon viel mehr interessante Varianten, als hier darstellbar ist. Gewiss fallen Ihnen dazu einige Beispiele ein. Aber: Niemand kann alles wissen. Sehen Sie es also bitte dieser Buchreihe nach, dass sie auf Erfahrungen und Kenntnissen beruht, die zwangsläufig begrenzt sind.

Wie kann das Spektrum der Mischprozesse erschlossen werden? Es gibt wegen der großen Vielfalt keine bequeme Top-Down-Zerlegung des Begriffs „Mischen", die sich als Kapiteleinteilung eignen würde, weil je nach aktueller Aufgabe und Anwendungskontext sowohl Standardlösungen als auch kombinierte Lösungen sinnvoll sein können. Daher erscheint es als sinnvoller, sich dem Thema aus mehreren Sichten zu nähern, die einander ergänzen. Die folgenden Kapitel stellen die Vielfalt der Mischprozesse an Beispielen vor und entwickeln die zugehörigen JSP-Modelle.

Es wäre jedoch eine langweilende Wiederholung des bereits bekannten Vorgehens, fänden die daraus abgeleiteten Implementierungen vollständig parallel in C und in Java® statt. Stattdessen hat der Autor sich dafür entschieden, die dateibasierten Mischverarbeitungen mit C zu implementieren, während die Java-Lösungen auf der Basis von MySQL aufbauen und wegen der damit möglichen Alternativen inhaltliche Abweichungen zu den traditionellen Ansätzen aufweisen dürfen.

1.3 Mischen: Ein Spektrum

Eine ganz einfache Frage ist am Beginn der Entwicklung einer Mischverarbeitung immer zu stellen:

Wie liegt das zu mischende Datenmaterial physisch vor?

- Ist es in sequentieller Form abgespeichert, als Speichertabelle oder einer anderen hauptspeicherorientierten Organisationsform, in irgendeiner dateibasierten Direktzugriffsform, in einer Datenbank?
- Wird es transaktionell gesendet, und wenn ja, einzeln oder zusammengefasst?
- Ist es in je einer physischen Informationseinheit zusammengefasst, oder sind mehrere physische Zugriffe nötig, um die Daten intern für jeden Misch-Schritt bereitzustellen (etwa aus XML-Strukturen, S.W.I.F.T.-Nachrichten)?
- Wie viele Datenquellen sind beteiligt? Zwei sind Standard, mehr kommen deutlich seltener vor.
- Wie umfangreich ist das Datenmaterial?

Der traditionelle Begriff des Mischens meint immer eine Verarbeitung sortierter Datenmengen gegeneinander. Diese Voraussetzung ist jedoch häufig nicht gegeben beziehungsweise nicht erforderlich. Die nächste Frage lautet daher:

Synchrones oder asynchrones Mischen?

- Besitzt das Datenmaterial die für das Mischen passende Sortierung (synchrones Mischen)? Aufsteigend oder absteigend?
- Gibt es Ranggruppenstrukturen – gegebenenfalls unterschiedlicher Tiefe – in diesem Material?
- Sind die für das Mischen relevanten Schlüssel identisch aufgebaut respektive kompatibel?
- Wenn keine passende Sortierung vorliegt (asynchrones Mischen), wie ist das Datenmaterial dann strukturiert – falls es überhaupt irgendwie geordnet ist?
- Muss diese Struktur erhalten bleiben oder müssen mehrere Strukturen nach einem vorgegebenen Prinzip verschmolzen werden?
- Eignen sich die Datenmengen, die in den Mischvorgang eingehen, überhaupt für asynchrones Mischen? Sind sie nicht zu groß?

Datenbanken, speziell relationale, ermöglichen es, Teilmengen zu mischen. Das hat den großen Vorteil, dass die beteiligten Bestände nicht vollständig gegeneinander verarbeitet werden müssen, sondern nur Ausschnitte davon. Damit verlassen wir das Feld der sequentiellen Verarbeitung und beziehen neue Gesichtspunkte ein, was die vorliegende Misch-Thematik komplexer macht. Allerdings bewegen wir uns so immer noch innerhalb des traditionellen Paradigmas, das von der Verarbeitung von Inhalten ausgeht, nicht aber von der Verarbeitung von Strukturen. Solche Strukturen liegen beispielsweise mit der Organisation von Dateien und Ordnern in File-Systemen vor. Auch in diesem Umfeld sind Mischvorgänge denkbar, etwa um Originalbestände und deren Kopien gegeneinander abzugleichen, wie es für Datensicherungen nötig ist. Wenn Differenzen zwischen solchen Beständen allein anhand äußerer Merkmale – als da sind: abweichende Strukturen, verschiedene Änderungsdatum-Stände oder unterschiedliche Längen – festgestellt werden können, müssen die beteiligten Dateien nicht geöffnet und verglichen werden, um die jeweils vorgesehenen Aktionen auszulösen.

Was folgt daraus für die Gliederung dieses offensichtlich umfangreichen Themas?

Die oben aufgelisteten Fragen und Aspekte sind sicherlich nicht sämtliche, die zum Thema „Mischen" gehören, aber doch schon ziemlich viele. Daher ist es nicht möglich, hier für alle Aufgabenstellungen die passenden Antworten anzubieten und Beispiele zu präsentieren. Das ist ja auch nicht die Aufgabe dieser Staffel. Vielmehr soll ein repräsentativer Ausschnitt der oben angesprochenen Themen behandelt und in den JSP-Kontext gesetzt werden. Was erwartet Sie da?

- Das nächste Kapitel geht von einer asynchronen Mischaufgabe aus, bietet eine erste – sehr unschöne – Lösung an und entwickelt eine alternative Lösungsmöglichkeit auf Direktzugriffs-Basis.
- Die anschließenden Kapitel stellen diverse Varianten des Ausgangsproblems vor, wie sie in der Praxis vorkommen, entwickeln strukturierte Lösungen auf der Basis von JSP/MSP, zeigen deren – sequentielle und relationale – Implementierungen, und verifizieren diese anhand von Tests.
- Danach geht es um das Mischen zweier sortierter sequentieller Bestände, also *den* Standardfall und seine Varianten. Dies ist auch der Ort, um das JOIN-Statement mit programmierten Lösungen zu vergleichen.
- Das Mischen multipler Datenmengen und das Mischen mit unterschiedlich vielen Rangstufen sorgt meistens für Verwirrung. Die letzten „traditionellen" Misch-Kapitel am Anfang des Bands 3-B2 klären darüber auf. Hier gehört auch das Problem mit den fehlenden oder überflüssigen Schul-Daten-Dreiergruppen hin.
- Völlig neu – aus der Perspektive herkömmlicher Mischverarbeitungen – ist das Mischen von Beständen, die in einem hierarchisch aufgebauten Dateisystem – oder einer ähnlichen Struktur – stehen und miteinander korrespondieren. Hier geht es zunächst um flache Strukturen, wie sie auf den Memory-Chips von Digitalkameras anzutreffen sind (sofern man sich auf Fotos beschränkt). Der letzte Schritt führt zu beliebigen Baumstrukturen, wie sie typischerweise beim Anlegen und Abgleichen von Backups oder Kopie-Beständen vorliegen.

1.4 Mischen auf relationaler Basis

Unbestreitbar wohnt der Entscheidung des Autors, nicht-traditionelle Mischprozesse ausschließlich auf relationaler Basis zu diskutieren, eine gewisse Willkür inne. Wo bleiben beispielsweise die objektrelationalen Systeme oder die älteren Konkurrenten wie etwa IMS DB® und ADABAS®? Bei Letzteren erscheint ihr Ausschluss allein wegen ihres vergleichsweise hohen Alters und ihrer abnehmenden Relevanz als gerechtfertigt, aber sollten Java-basierte Implementierungen nicht gleich den Schritt zu objektrelationalen Datenbanken tun?

Wie so oft in dieser Buchreihe hat diese Entscheidung rein pragmatische Gründe. Es geht darum, dass die Leserinnen und Leser selbst mit den Beispielen arbeiten können, wobei es wesentlich darauf ankommt, ob die verwendete Technologie allgemein verfügbar und kostenlos ist. Das trifft hier für mehrere relationale Datenbanksysteme zu, die auf PCs, Apple®-Systemen oder anderer weit verbreiteter Hardware operieren. Unter diesen Systemen fiel die Wahl auf MySQL, weil es a) ein client-server-basiertes und damit technologietypisches Datenbanksystem ist, b) mit „Strong Typing" arbeitet, was ein diszipliniertes Arbeiten erzwingt und auch für die „großen Brüder" auf Hochleistungssystemen üblich ist, c) das relationale Leistungsspektrum exzellent abdeckt und d) über zwei visuelle Schnittstellen verfügt, die beide nützlich sind.

Wie stets, wenn zwei Themen sich zueinander orthogonal verhalten, also keines der beiden unter dem anderen hierarchisch eingeordnet werden kann, kommt es zu einem Gliederungsproblem: Sollte das Thema „Mischen" vorerst auf sequentielle Bestände begrenzt werden, um es später, nach der eingehenden Diskussion datenbankorientierter Themen – im dritten Teil der dritten Staffel – erneut aufzugreifen und auf relationale Bestände anzuwenden? Dazu müssten die Mischaufgaben zumindest erneut angerissen werden, und es wäre nicht ohne weiteres möglich, die Implementierungsalternativen miteinander zu vergleichen. Anders herum können diese Alternativen zwar thematisch mit den „traditionellen" Optionen zusammengefasst werden, aber

nur um den Preis, auf die relationalen Grundlagenkapitel vorerst zu verzichten. Der Autor hat sich für diesen zweiten Weg entschieden, weil mit relationalen Beständen schon gearbeitet werden kann, wenn hierüber wenig Grundlagenwissen vorhanden ist. Das ist ein umfangreicher Vorgriff auf Kommendes, ganz klar, aber er erspart öde Wiederholungen. Diese Strukturierung liegt schon dem vorhergehenden Unterkapitel zugrunde; hier wird daher nur nachgereicht, warum das so ist.

1.5 Entflechten: Gelbe und weiße Karten

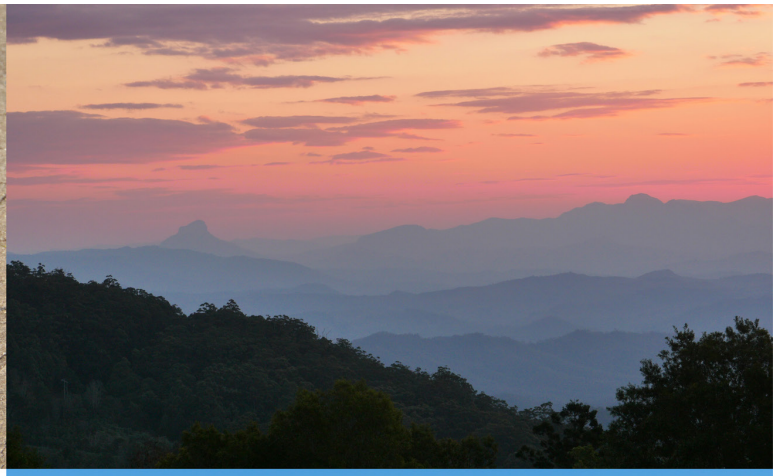
Diese kryptische Überschrift muss natürlich erklärt werden… Lang, lang ist's her, dass der Autor einen Kursus in Treiberprogrammierung für die TR86® – den Ein- und Ausgabe-Prozessrechner des TR440®-Großrechners – bei Telefunken in Konstanz besuchte. Wohl kaum jemand weiß noch heute, dass Telefunken sich damals auch als Rechnerhersteller einen großen Namen machte, aber letztlich gegen die damals neuen IBM®-Systeme und andere Konkurrenten unterlag. Eine Kursus-Aufgabe bestand jedenfalls darin, die verstreut in einem ansonsten weißen Kartenstapel steckenden gelben Karten mit Hilfe eines Lochkartenleser auszusortieren. Diese Leser waren beeindruckend große, metallisch glänzende Geräte mit langen, horizontalen, nach oben offenen Kartenschächten. Eine Aufgabe der weißbekittelten Operateure, wie die Maschinenbediener damals hießen, bestand darin, die von den Benutzern angelieferten Kartenstapel in die Leser-Eingabeschächte zu laden und dann einen Start-Knopf zu betätigen. Danach „fraß" der jeweilige Leser mit rasender Geschwindigkeit diese Kartenstapel und entließ sie auf seiner Ausgangsseite in einen ebenso langen Kartenschacht. Von dort entnahmen die Operateure die Karten und machten entweder ein Gummibändchen um schmale Stapel, oder sie stellten längere Stapel in diejenigen Boxen zurück, in denen sie angekommen waren. Um das Trennen der Stapel für die Operateure nicht zu mühsam zu machen, hatte die erste Karte eines Stapels eine andere Farbe als die – meistens weißen – nachfolgenden Karten.

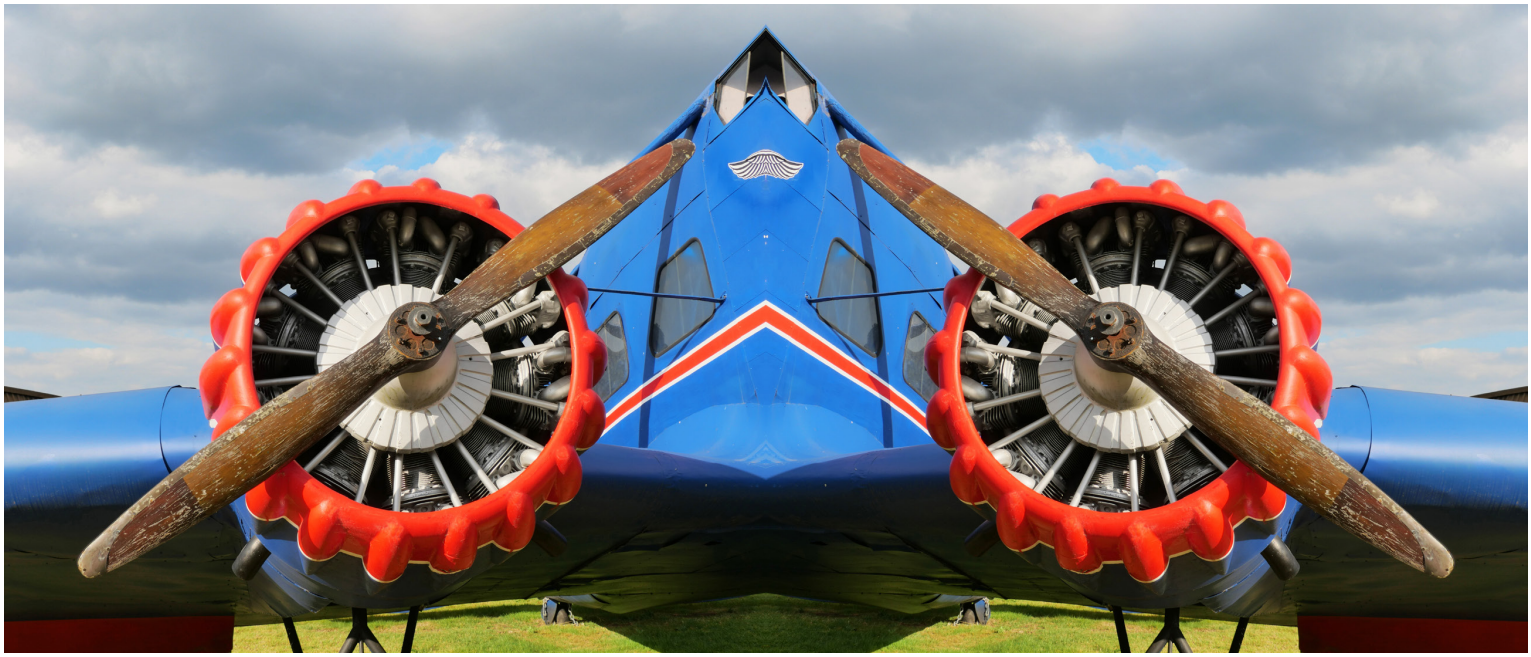
Den Kartenlesern waren die Farben der Karten völlig egal. Sie waren aber imstande, defekte Karten oder solche, deren Lochung nicht lesbar war, in ein Aussteuerfach zu schießen, das nur einen kurzen Stapel aufzunehmen vermochte. Gewöhnlich hielt der Leser an, sobald eine solche Karte entdeckt wurde, aber das war eine in seinem Treiber programmierte Reaktion, kein Muss.

Zurück zur Aufgabe: Um die gelben und die weißen Karten voneinander unterscheiden zu können, waren sie unterschiedlich gelocht. Der zu programmierende Treiber musste daher nach jedem Interrupt vom Leser die Lochung der aktuell eingelesenen Karte prüfen, um zu entscheiden, ob sie „weiß" war und geradeaus weiterlaufen durfte, oder ob es sich um eine „gelbe" Karte handelte, für die ein Aussteuerungsbefehl an den Leser zu geben war. Für Laien sah der Lesevorgang nach dem Einlegen des gemischten Stapels und dem Druck auf den Startknopf tatsächlich so aus, als ob der Kartenleser imstande wäre, auf die Farben der Karten zu reagieren: Nur weiße Karten kamen hinten aus dem Leser wieder heraus, und nur gelbe sammelten sich im Aussteuerungsschacht.

Was hat das alles mit dem Entflechten als Umkehroperation des Mischens zu tun? *Dieses* Entflechten dient dazu, gemeinsam eintreffende Daten gleicher Beschaffenheit in getrennte Ausgabeeinrichtungen oder andere Kanäle zu leiten, wobei davor gegebenenfalls eine mehr oder minder komplexe Verarbeitung stattfindet. Damit das möglich ist, müssen diese Daten über Unterscheidungsmerkmale verfügen, anhand derer über ihre Zuordnung entschieden werden kann. Hierfür sind viele Beispiele denkbar, aber die weißen und gelben Karten sind durchaus repräsentativ. Im Gegensatz zum Mischen ist das Entflechten von weit geringerer Komplexität und wird daher in den kommenden Kapiteln allenfalls am Rand behandelt.

Der Begriff des Entflechtens wird im JSP noch in einem ganz anderen Sinn verwendet, nämlich der Lösung von Korrespondenzproblemen mithilfe der Pufferung von Daten im Hauptspeicher, kurz „Entflechten über den Hauptspeicher" genannt. In diesem Kontext ist *jenes* Entflechten eher symbolisch gemeint, also als „Entknoten verknoteter Daten". Das Durcheinander der Begriffe lässt sich also noch vergrößern…





09/2019 oben: Impressionen vom O'Reillys Rainforest Retreat (Queensland, Australien)
unten: Fotomontage anhand des Propellerflugzeugs

Mischen: (im)possible

Den geschätzten Leserinnen und Lesern mag das im folgenden geschilderte sequentielle Mischverfahren genauso absurd wie die Fotomontage auf der linken Seite unten vorkommen. Es funktioniert teilweise so, als würde der linke Propeller sich nur mit einem kleinen Bruchteil der Rotationsgeschwindigkeit des rechten Propellers bewegen, damit zu jeder linken Propellerposition eine volle Umdrehung des rechten Propellers gehört. Wenn aber gar nichts anderes machbar ist, dann kann man so zwar nicht fliegen, aber ein funktionierendes – extrem ineffizientes – Mischverfahren entwerfen. So wird zwar hoffentlich niemand vorgehen, aber es kann nicht schaden, sich ein wenig zu gruseln.

Dagegen kann dieselbe Aufgabe elegant mit Direktzugriffsmethoden gelöst werden. Um im Bild zu bleiben: Die Papageien auf der linken Seite picken gezielt ihr Futter auf. Sie können die Futterkörner auf Anhieb von kleinen Steinchen unterscheiden.

Im Zeitalter der Datenbank- und Transaktionssysteme kann es des öfteren nicht garantiert werden, dass die zu verarbeitenden Datenmengen tatsächlich sortiert vorliegen beziehungsweise vom Mischprogramm mit vertretbarem Aufwand sortiert eingelesen werden können, was zweierlei ist. Es ist daher ratsam, die bisher übliche starre Erwartungshaltung abzulegen und darüber nachzudenken, ob und wie zwei oder mehr Datenmengen ohne eine zugesicherte Sortierung gemischt werden können. Der Einfachheit halber beginnen wir mit zwei – extern gespeicherten – Eingabebeständen, deren Inhalte zu einem gemeinsamen Ausgabebestand zusammengemischt werden sollen. Die folgende Grafik zeigt diese Aufgabe:

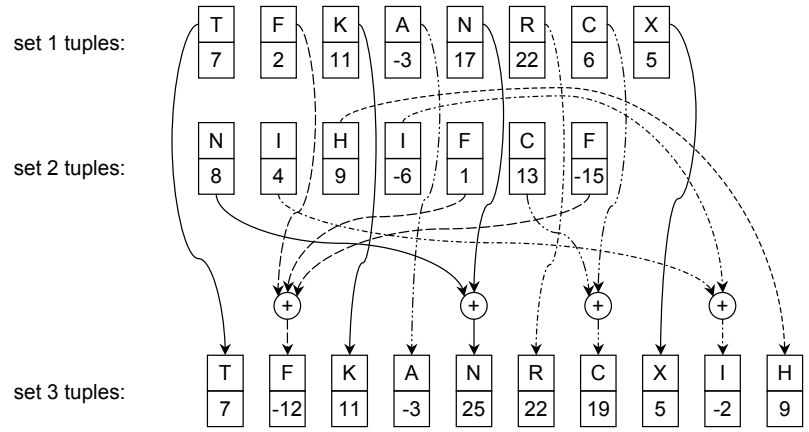


Abbildung 2.1: *Mischen von Eingabe-Beständen mit Schlüsselduplikaten im Tuple Set 2*

Was bedeutet diese Grafik?

- Jede Datenmenge – Tuple Set genannt – enthält Daten im gleichen Format. Die Buchstaben stehen für beliebige, gleich aufgebaute Schlüssel. Die Zahlen können nur Werte des laut Format zulässigen Ganzzahl-Bereichs annehmen, bei denen Überlauf und Unterlauf während der gezeigten Summationen ausgeschlossen sind.

- Aus den beiden Ausgangsmengen T1 und T2 soll die Zielmenge T3 gebildet werden. Die Reihenfolge der Schlüssel/Zahl-Tupel in allen drei Mengen ist beliebig. T1 und T3 dürfen jeden Schlüssel nur einmal enthalten, T2 mehrfach. Die Anzahlen der Tupel sind nicht nach oben begrenzt. Die Grafik enthält nur aus Gründen der Übersichtlichkeit so wenige Tupel.
- Wenn Schlüssel zwischen T1 und T2 übereinstimmen, ist die Summe der Zahlen in den betreffenden Tupeln zu bilden und ein neues Tupel Schlüssel/Summe als T3-Element abzulegen. Fälle: `'F'` | `'N'` | `'C'`
- Sind Schlüssel nur in T1 vorhanden, werden diese Tupel unverändert nach T3 kopiert. Fälle: `'T'` | `'K'` | `'A'` | `'R'` | `'X'`
- Schlüssel, die nur in T2 vorkommen, können einmalig oder mehrfach auftreten. Tupel mit einmalig auftretenden Schlüsseln werden ebenfalls nach T3 kopiert. Fall: `'H'`. Andernfalls werden die Zahlen addiert und das jeweilige Tupel Schlüssel/Summe wird als T3-Element abgelegt. Fall: `'I'`
- Es gibt keine Vorschrift über die zeitliche Abfolge der Misch-Schritte. Zwischenzustände sind zugelassen. So kann zum Beispiel ein T3-Zwischenzustand als neue Menge T1 dienen, solange die Aufgabe korrekt gelöst wird.
- Die physischen Organisationsformen für T1, T2 und T3 sind nicht definiert. Es muss lediglich möglich sein, die Daten aus T1 und T2 sequentiell zu lesen.

2.1 Erster Modellierungsversuch

Spontan liegt die als „Abbildung 2.2: Intuitives Modell der Ein- und Ausgabedatenstrukturen“ auf Seite 11 folgende Modellierung der beteiligten Datenstrukturen nahe.

Die zentrale Idee in dieser Grafik ist, zwischen zueinandergehörenden und isolierten T1- / T2-Tupeln zu unterscheiden. Da T1 keine Schlüsselduplikate enthalten kann, sind darin nur isolierte Tupel (Fälle: `'T'` | `'K'` | `'A'` | `'R'` | `'X'`) und solche möglich, die mit einem T2-Tupel oder mehreren T2-Tupeln durch identische Schlüssel verknüpft sind (Fälle: `'F'` | `'N'` | `'C'`). In T2 kann es dagegen isolierte Tupel (Fall: `'H'`), mit T1 durch identische Schlüssel verknüpfte Tupel (Fälle: `'F'` | `'N'` | `'C'`) und solche Tupel geben, die nur in T2 mehrfach vorkommen (Fall: `'I'`).

Die gestrichelten Korrespondenzlinien zeigen, welche Tupel zusammengehören. Allerdings existieren nur für die isolierten Tupel gültige Korrespondenzen wegen gleicher Anzahl und Reihenfolge. Bei den Tupeln, die in die Summationen eingehen, stimmen die Reihenfolgen nicht, weil es keine Sortierung und folglich keine Gruppierung gibt. Es sind beliebig große Lücken zwischen je zwei zusammengehörenden Tupeln möglich. Wenn das Programm zu jeder Zeit nur je ein T1-Tupel und ein T2-Tupel im Zugriff hat, was bei sequentieller Verarbeitung ohne Pufferung zutrifft, existiert keine gleichzeitige Verarbeitbarkeit, die erforderlich wäre, um die T3-Tupel in einem einzigen Durchgang zu erzeugen. Daher sind diese Linien mit Fragezeichen versehen.

Wie sieht es mit den Kontrollen aus?

I1	T1-Ende
I2	T2-Ende
S3	T1-Tupel-Key = T2-Tupel-Key
S4	T1-Tupel-Key = T2-Tupel-Key | T2-Tupel-Key x = T2-Tupel-Key y | sonst
I5	T1-Ende und T2-Ende
S6	T1-Tupel-Key = T2-Tupel-Key | T2-Tupel-Key x = T2-Tupel-Key y | Isoliertes T1-Tupel | sonst

Die Auswahlbedingungen S2, S4 und – davon abgeleitet – S6 sind allerdings nicht entscheidbar, weil die dafür relevanten Tupel nicht gleichzeitig auf Schlüsselgleichheit untersucht werden können.

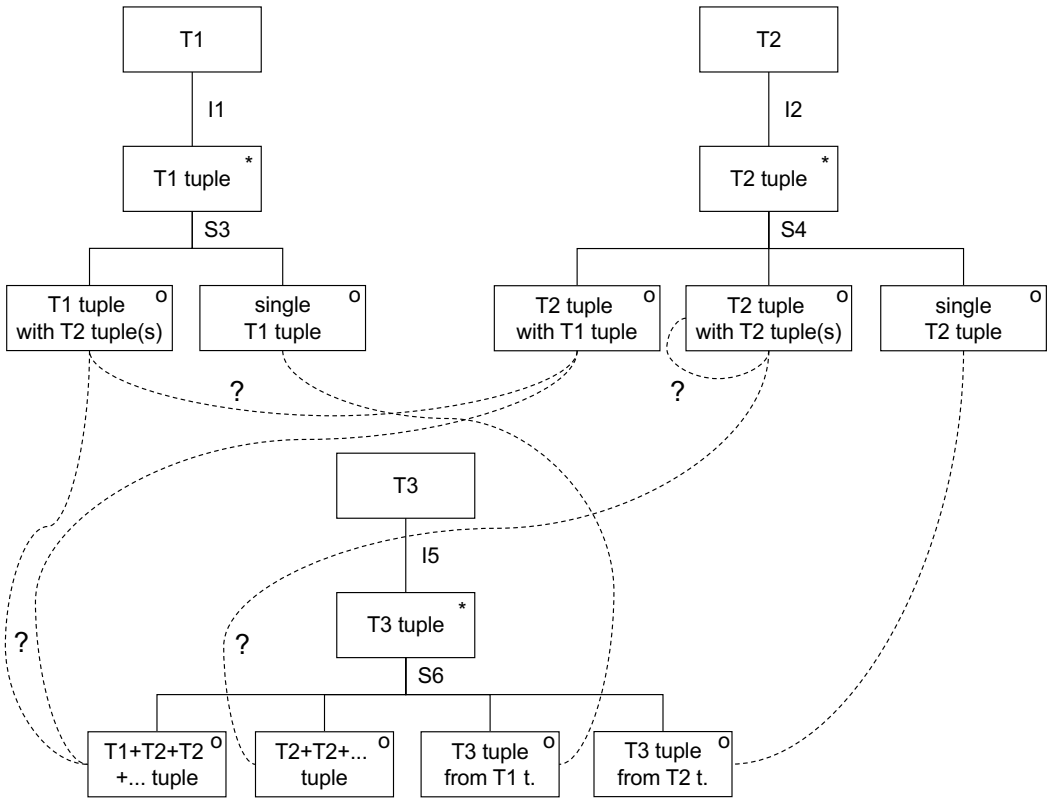


Abbildung 2.2: *Intuitives Modell der Ein- und Ausgabedatenstrukturen*

Programmieraufgaben wie die oben dargestellte, bei denen aus zwei oder mehr unsortierten beziehungsweise unpassend sortierten Mengen eine Ergebnismenge erzeugt werden muss, treten in der Praxis häufig auf. Beispiele:

a) Zusammenfassung von Stücklisten eines zusammengesetzten Objekts, zum Beispiel eines Schiffs
b) Aggregation von Wertpapierhandelsaufträgen vor der Weitergabe an einen Börsenmakler
c) Addition von Inventurmengen , die von mehreren Inventurgruppen in einem großen Lager mit chaotischer Lagerung – das heißt ohne feste Platzzuordnungen – ermittelt wurden

Es liegt dann zwar möglicherweise eine – vollständige oder teilweise – Sortiertheit hinsichtlich irgendwelcher Schlüssel vor, nur ist diese im Rahmen der Aufgabe gegebenenfalls nicht nutzbar. Stücklisten sind sachlogisch aufgebaut, nicht schlüsselsequentiell. Wertpapierhandelsaufträge betreffen Aktien oder andere Investmentpapiere, die jeweils anhand ihrer Wertpapierkennummer (WKN) oder ISIN (International Security Identification Number) bekannt sind. Die Aufträge sind aber möglicherweise nach dem FIFO-Prinzip – first in, first out – anstatt nach WKN oder ISIN sortiert abgelegt. Inventurlisten orientieren sich an Warengruppen oder Lagerzonen, nicht an Warenidentifikationen (zum Beispiel EAN-Codes).

Welche typischen Varianten zur Lösung solcher Aufgaben gibt es?

2.2 Rohe Gewalt

Eine rein sequentielle Lösung, bei der T1 und T2 iterativ – phasenweise – gegeneinander verarbeitet werden, ist im Prinzip möglich, aber aufwendig. Für jedes T1-Tupel müssen zunächst sämtliche T2-Tupel durchgelesen werden, um übereinstimmende Schlüssel zu finden. T1-Tupel mit T2-Treffer(n) ergeben T3-Summentupel. T1-Tupel ohne T2-Treffer – isolierte T1-Tupel – werden unverändert nach T3 geschrieben. Dafür sind folglich n * m Lesezugriffe erforderlich, wobei n die Anzahl der T1-Tupel und m die Anzahl der T2-Tupel ist. Die folgende Grafik zeigt den ersten Durchgang der Verarbeitung am Beispiel:

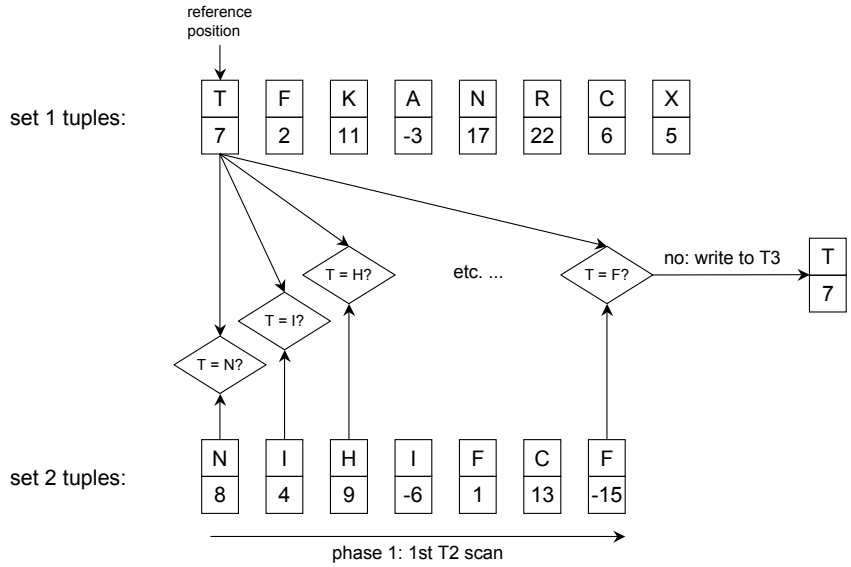


Abbildung 2.3: *Erster Scan durch T2 anhand des T1-Schlüssels* `'T'` *und Schreiben nach „Mismatch“*

Der zweite Durchgang erzielt zwei T2-Treffer, wie die „Abbildung 2.4: Zweiter Scan durch T2 anhand des T1-Schlüssels 'F', Kumulieren und Schreiben“ auf Seite 13 zeigt… und so weiter und so fort. So entsteht die Menge T3 bis auf die Fälle `'I'` und `'H'`. Letztere sind allerdings nicht leicht identifizierbar, weil nur noch T2-interne Paare respektive Mehrlinge und isolierte T2-Tupel verwendet werden dürfen. Diese können durch einen Abgleich gegen T1 allein nicht gefunden werden, weil für jedes aktuell im Zugriff stehende T2-Tupel hierdurch nur festgestellt werden kann, ob ein zu ihm paariges T1-Tupel existiert. Diese Prüfung kostet maximal weitere n * m Lesezugriffe, weil sie für das geprüfte T2-Tupel – auf der Referenzposition in T2 – bei einem T1-Treffer sofort abgebrochen werden kann. Das zeigt die „Abbildung 2.5: Erster Scan durch T1 anhand des T2-Schlüssels 'N': Schlüssel ist bereits bekannt“ auf Seite 13.

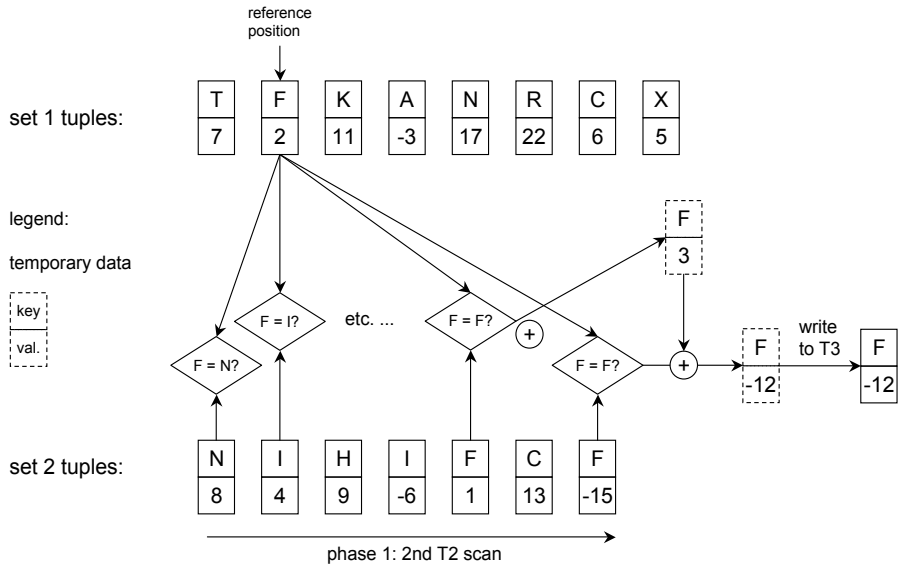


Abbildung 2.4: *Zweiter Scan durch T2 anhand des T1-Schlüssels* `'F'`*, Kumulieren und Schreiben*

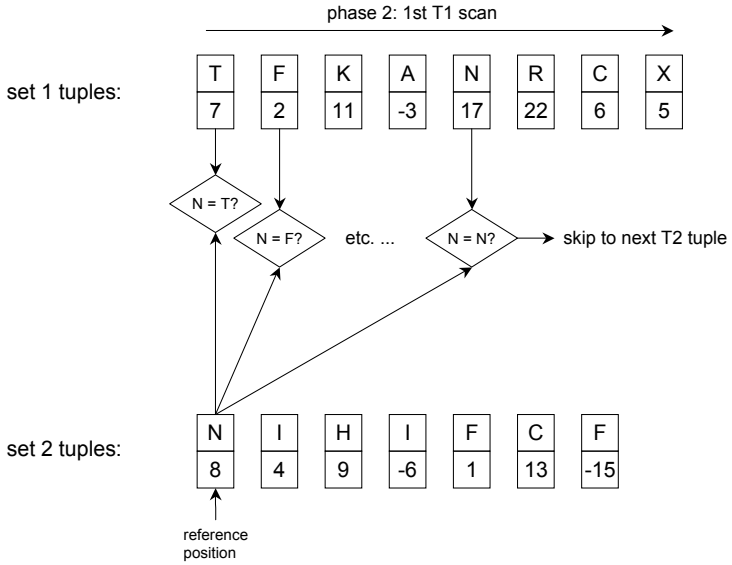


Abbildung 2.5: *Erster Scan durch T1 anhand des T2-Schlüssels* `'N'`*: Schlüssel ist bereits bekannt*

Um festzustellen, ob es mindestens ein weiteres Nur-T2-Tupel mit demselben Schlüssel gibt, muss T2 folglich zusätzlich mit sich selbst verglichen werden, was maximal m * m Lesezugriffe erfordert, wenn kein T2-Tupel in T1 vorkommt. Tritt ein Tupel auf beiden T2-Seiten nur in derselben Position auf, so handelt es sich um ein isoliertes Tupel, das nach T3 ausgegeben werden darf. Kommt es aber mehrfach vor, so wird ein weiteres T3-Summentupel gebildet und ausgegeben. Mehrfachausgaben derselben Summentupel verhindert die Regel, dass die Positionen von gegenüberliegenden paarigen T2-Tupeln höher als die des T2-Tupels der Referenzposition sein müssen. Ist die Position eines Gegenseite-T2-Tupels kleiner, so wurde das aktuelle T2-

Tupel bereits berücksichtigt und es kann sofort auf das nächste T2-Tupel fortgeschaltet werden. Daher werden im Allgemeinen weit weniger als m * m Lesezugriffe benötigt.

Die folgende Grafik zeigt, wie das mehrfach vorkommende Tupel mit dem Schlüssel `'I'` beim zweiten Durchgang nach erfolglosem Vergleich mit T1 – nicht gezeigt – verarbeitet wird:

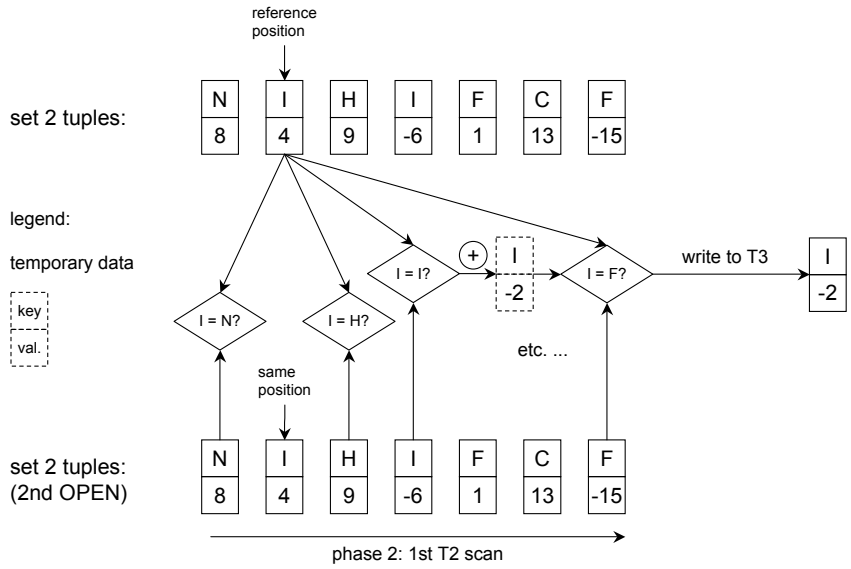


Abbildung 2.6: *T2-gegen-T2-Scan für den Schlüssel `'I'`, Kumulieren und Schreiben*

Für den Schlüssel `'H'` gibt es keinen Treffer außer auf derselben T2-Position; das isolierte H-Tupel wird daher nach dem zweiten Durchlauf gegenüber T2 nach T3 geschrieben. Dafür ist keine eigene Grafik nötig. Nun bleibt noch der Fall des zweiten I-Tupels übrig:

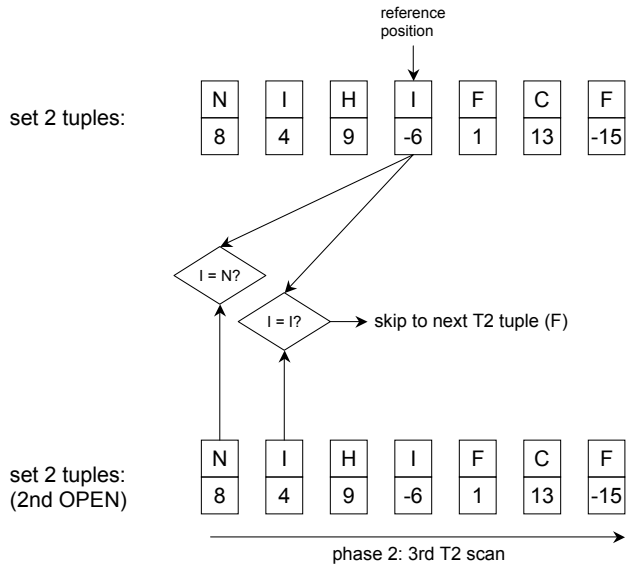


Abbildung 2.7: *DritterT2-gegen-T2-Scan für den vierten Schlüssel `'I'`: Schlüssel ist bereits bekannt*

Die Tupel-Position kann durch einfaches Abzählen bestimmt werden. In der obigen Grafik ist die Reference Position gleich 4 und die Position des Treffers gleich 2; somit endet der Vergleichsdurchgang sofort. Die Gesamtzahl der Lesezugriffe und Vergleiche ist demnach jeweils maximal m * (2 * n + m). Bei 100 T1-Tupeln und 10 T2-Tupeln sind das schon 10 * (200 + 10) = 2100! Mit synchronem Mischen genügt dagegen ein einziger Durchlauf mit n + m Lesezugriffen und Vergleichen. „Rohe Gewalt" ist daher – wie meistens – völlig indiskutabel und wird nicht weiterverfolgt. Auch Lösungen mit Zwischendateien sind ineffizient und kompliziert. Selbst wenn beide Tupelmengen in den Hauptspeicher passen, kann das Verfahren nicht empfohlen werden.

Wie kann die vorliegende Aufgabe dennoch sinnvoll gelöst werden?

2.3 Entflechten über den Hauptspeicher

Hier ist der Begriff „Entflechten" eher als Verlegenheitsbezeichnung dafür zu verstehen, dass es nicht leicht ist, auf rein sequentieller Basis eine akzeptable Lösung zu finden. Durch die Nutzung des Hauptspeichers als Puffer ergeben sich aber erheblich größere Freiheitsgrade für die Lösungsfindung. Bei kleinen Mengen T1 oder T2 ist es vorstellbar, eine davon oder beide in den Hauptspeicher des Mischprogramms zu laden und die Menge T3 anhand dieser gepufferten Daten zu produzieren. Wenn T2 in den Hauptspeicher passt, T1 aber nicht – welche Lösungen bieten sich an?

- Das Mischen erfolgt durch iterative Verarbeitung der beiden Mengen gegeneinander, ohne etwas an deren Reihenfolgen zu ändern.
- T2 wird vorverdichtet, indem die Zahlen in Tupeln mit gleichen Schlüsseln aufaddiert werden. Anschliessend erfolgt das Mischen gegen T1.
- T2 wird zusätzlich sortiert, um als Lookup-Tabelle für das binäre Suchen zu dienen.

Diese Lösungsvarianten werden in den Folgekapiteln modelliert und jeweils bis zur Programmstruktur mit Operationen und Kontrollen ausformuliert. Das Vorhandensein einer Sortierfunktion wird dabei vorausgesetzt.

2.4 Synchrones Mischen im Hauptspeicher

Ohne externe Sortierung, zum Beispiel mit einem Dienstprogramm, ist das Reißverschlußverfahren nur dann anwendbar, wenn T1 und T2 beide in den Hauptspeicher passen. Ausgehend von der zuletzt genannten Lösungsvariante muss also auch T1 komplett eingelesen werden. Diese Form des Sortierens und Mischens im Hauptspeicher ist technisch die vorteilhafteste, weil externes Sortieren bei kleinen Mengen wesentlich mehr Betriebsmittel benötigt. Auch diese Lösung wird später in diesem Band fertig modelliert.

In den nachfolgenden Kapiteln sind die Eingabebestände bezüglich der Mischaufgabe jedoch unsortiert, auch wenn sie irgendeine andere – für die Aufgabe nicht brauchbare – Sortierfolge aufweisen. Die nachfolgende Option steht daher zunächst nicht zur Verfügung:

2.5 Synchrones Mischen sortierter Bestände

Diese Mischtechnik war in Zeiten kleiner Hauptspeicher und sequentiell aufgebauter Datenbestände – meistens auf Bändern – die einzige gangbare Lösung. Das bedeutet aber nicht, dass sie heute überflüssig geworden ist. Auch mit großen Hauptspeichern und schnellen Direktzugriffsspeichern – Disk Drives oder Solid State Devices = Halbleiterspeicher auf Flash-Technik-Basis als Platten-Emulation – gibt es vor allem bei sehr umfangreichen Beständen keine konkurrenzfähige Alternative zum Reißverschlussverfahren. Allerdings ist die Palette sinnvoller Anwendungen klein geworden. Viele traditionell unvermeidbare Mischverarbeitungen wurden durch Datenbanktechniken und JOINs abgelöst. Für Standard-Mischvorgänge existieren Dienstprogramme.

Interessanterweise werden immer noch neue Anwendungsmöglichkeiten für dieses traditionelle Verfahren entdeckt. So berichteten – 2014 in *Spektrum der Wissenschaft* – Forscher, die fossile DNA-Sequenzen gegen DNA-Datenbanken abgleichen, sie hätten die Laufzeit ihres Programms dadurch radikal vermindert, indem sie die digital codierten Sequenzen auf beiden Seiten aufsteigend sortierten und dann gegeneinander mischten.

Sonstige Nicht-Standard-Fälle repräsentieren eine breite Variantenfülle von Aufgabenstellungen, die sich in zwei Dimensionen erstreckt, nämlich einerseits der Anzahl-Dimension der zu mischenden Bestände und andererseits der Rangschlüssel-Dimension, das heißt der Konsequenzen gegebenenfalls unterschiedlicher Rangschlüsseltiefen. Beide Dimensionen können kombiniert auftreten, was nach Anstrengungen verlangt, die dadurch ausgelöste exponentielle Fall-Vervielfachung in den Griff zu bekommen.

Dieses umfangreiche Thema kann auch aus Platzgründen erst später behandelt werden.

2.6 Alternative: Direktzugriffe

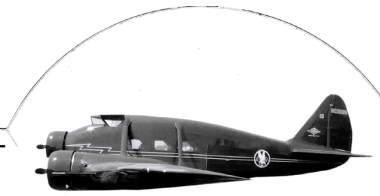
Auf sequentieller Basis und ohne Sortierung ist das eingangs geschilderte Problem zwar lösbar, aber nur in ganz extremen Fällen wird man zu solchen „Klimmzügen" greifen müssen und wollen. Immerhin machen die vorangehenden Überlegungen deutlich, mit welch unangenehmen Fragestellungen die Entwickler(innen) konfrontiert waren, als die Maschinen noch klein waren und nur sequentielle Medien, zum Beispiel Banddateien, verarbeiten konnten. Kein Wunder, dass elegante und effiziente Sortierverfahren rasch eine hohe Relevanz erreichten.

Als die ersten Direktzugriffsmethoden auf der Basis rotierender Medien möglich wurden, änderte sich vieles, wenn nicht alles. Plötzlich entfielen die sequentiellen Zwänge, um die das Denken der Programmierer(innen) bis dahin kreiste, und es tat sich eine neue Welt auf – so schien es zumindest. Frederick Brooks schildert in [8] auf der Seite 99 eindrucksvoll, wie die Verfügbarkeit schneller Direktzugriffsspeicher die Entwickler wegen der unvermeidlichen Platzrestriktionen im Hauptspeicher dazu verführte, ihre Programme in immer kleinere Module zu zerlegen, die bei Bedarf (nach)geladen wurden – und welch katastrophalen Effekt das für den Durchsatz hatte: Die Übersetzung eines FORTRAN-Programms benötigte auf einem Rechnermodell mit Trommelspeicher eine volle Minute für fünf Anweisungen! Dazu muss man wissen, dass Trommelspeicher weit schneller als die damaligen Plattenspeicher waren, weil jede Spur einer Magnettrommel einen eigenen Schreib-/Lesekopf hatte und somit die Latenzzeiten durch das Positionieren des Zugriffsarms entfielen.

Heutzutage, Jahrzehnte später, lächeln wir nur müde über solche K(r)ämpfe, bevor wir uns daran machen, die miserable Performance eines Programms zu untersuchen, das auf diversen relationalen Tabellen operiert. Das Problem des schlechten Umgangs mit den Ressourcen ist uns erhalten geblieben, nur dass die Mengen inzwischen ganz andere sind. Immerhin, und das ist eine gute Nachricht, kann das fatale Mischproblem mit unsortierten Tupeln mittels Direktzugriffen problemlos gelöst werden. Das soll im folgenden Kapitel gezeigt werden.

2.7 Nachtrag

Das am O'Reillys Rainforest Retreat ausgestellte Flugzeug ist ein Nachbau einer Stinson, der bis 2003 in einem Museum ausgestellt war und dann von der Familie O'Reilly erworben wurde. Er erinnert an die Rettung zweier Passagiere eines abgestürzten Stinson-Flugzeugs durch Bernard O'Reilly im Jahr 1937. Die Bildzusammenstellung am Kapitelanfang zeigt dieses Flugzeug rechts unten. Das nachfolgende Foto stammt von einer Tafel, die neben dem Flugzeug aufgestellt ist.



STINSON REPLICA

During a stormy afternoon on 19 February 1937 Stinson airliner VH-UHH, on route from Brisbane to Sydney, crashed into the McPherson Ranges killing four of the seven on board, sparking a search that failed to turn up any wreckage. Seven days after the crash Bernard O'Reilly set out on foot to find the Stinson wreckage, which he believed had gone down in Lamington National Park. Through his superior bush skills he found the crash site and to his surprise, two survivors in desperate need of medical attention. After boiling the billy and ensuring the survivors were as comfortable as possible, Bernard left them to organise a rescue party. He returned early the following morning with a doctor and an army of local farmers who stretchered the men to safety, 11 days after the crash.

The Model Plane

This model is the original replica of the Stinson airliner used in the 1987 film "Riddle of the Stinson" starring Jack Thompson and Richard Roxburgh. It stands at an impressive 4 metres high, 13 metres long and has a wing span of over 20 metres. The aircraft was rescued from the Wangaratta Air Museum in 2003 prior to the council closing the historical facility down and was purchased by the O'Reilly family and relocated to the guesthouse in 2012 for the 75th anniversary of the Stinson crash.

Bernard O'Reilly gives a personal recount of his search for the Stinson in his book 'Green Mountains' which is available for purchase at the Giftshop.

Proud and Binstead with members of the rescue party

Bernard O'Reilly enjoys a cup of tea after the rescue

Newspaper clipping 3 days after the rescue

O'Reilly's

06/2022 Seerose im botanischen Garten von Palermo (Sizilien)

Mischen per Direktzugriff

Seerosen und einige andere Wasserpflanzen haben im Verlauf der Evolution Blätter entwickelt, die Luftkammern enthalten und somit schwimmen können. Auf dieser Basis entstanden Blüten, deren erstaunliche Vielfalt und Schönheit wohl jede Betrachterin und jeden Betrachter berührt. Die Seerosen haben mit dieser Innovation eine große ökologische Nische erobert. Leider haben wir Informatiker bestenfalls Geistesblüten zu bieten, die wir mangels Anschauung nur intellektuell als attraktiv empfinden können. Dennoch sollten wir von den Seerosen lernen, wie sich das Nützliche (die Luftkammern) mit dem Schönen (der Blüten) verbinden lässt, wenn es zugleich gelingt, die ausgetretenen Pfade des Erwartbaren zu verlassen und den Sprung in eine vollständig andersartig gestaltete Alternative zu wagen. Das nun folgende Kapitel trägt die dafür nötigen Entwurfsaspekte zusammen und präsentiert eine Lösung für das im vorgehenden Kapitel präsentierte, sequentiell quasi unlösbare Problem.

Um das Versprechen aus dem Kapitel „2.6 Alternative: Direktzugriffe“ auf Seite 16 einzulösen, können und dürfen wir hier die lange Historie der vielfältigen Direktzugriffsmethoden einfach überspringen und stattdessen gleich auf diejenigen Direktzugriffe zu sprechen kommen, die mittels hash-organisierter oder indexbasierter relationaler Tabellen realisierbar sind.

3.1 Relationale Tabellen und Datenbanksysteme

… als Direktzugriffsalternativen können hier nicht ausführlich erklärt werden, denn das würde den Rahmen des Themas „Mischen“ komplett sprengen. Der Band 5-C2 geht darauf ausführlich ein. Er zeigt, wie die datenorientierte Methode und das relationale Paradigma miteinander verknüpft werden können, um moderne Anwendungssysteme zu konzipieren und realisieren. Daher kann hier nur das zusammengefasst werden, was für die Lösung des Mischproblems aktuell benötigt wird (und, wie üblich noch ein bisschen dazu…).

3.2 Relationale Tabellen

… sind Tabellen, die zueinander in Beziehung = Relation stehen. So können beispielsweise die Tupelmengen als sogenannte Key-Value-Tabellen aufgefasst werden, die durch die Mischaufgabe aufeinander bezogen sind:

TS1-Key	TS1-Value
T	+07
F	+02
K	+11
A	-03
N	+17
R	+22
C	+06
X	+05

TS2-Key	TS2-Value
N	+08
I	+04
H	+09
I	-06
F	+01
C	+13
F	-15

TS3-Key	TS3-Value
T	+07
F	-12
K	+11
A	-03
N	+25
R	+22
C	+19
X	+05
I	-02
H	+09

Die Grafiken im Kapitel „2 Mischen: (im)possible“ auf Seite 9 ff können nun anhand der Tabellendarstellung leicht nachvollzogen werden: Die TS3-Zeilen bis zum Key X sind teilweise Kopien aus TS1 und teilweise durch Verrechnung der korrespondierenden TS1-TS2-Zeilen entstanden. Die beiden letzten TS3-Zeilen stammen aus TS2, wobei die I-Zeile zwei TS2-Tupel zusammenfasst.

Es fällt auf, dass die Tabellenspalten Überschriften haben. Das ist ein Novum gegenüber sequentiellen Beständen, in denen die Daten ihre Benennung nicht mit sich führen. Man muss wissen, auf welchen Positionen eines sequentiellen Bestands sich welche Datenelemente befinden und wie diese dargestellt sind, um die richtigen Schlüsse für die Programmierung ziehen zu können. In relationalen Tabellen besitzen die Spalten dagegen nicht nur Namen, sondern auch Datentypen. Die obigen Beispieltabellen haben gemeinsam, dass die Key-Spalten einstellige Strings sind und die Value-Spalten vorzeichenbehaftete Integer-Zahlen in externer Darstellung zeigen. Die interne Ablageform solcher Daten kann durchaus stark von der Anschauung abweichen, aber das ist nur insofern wichtig, als es sich auf die Präsentation der Inhalte gegenüber zugreifenden Programmen auswirkt. Die Visualisierung durch einen „Tabellen-Browser“ verbirgt die reale Speicherungsform, während beispielsweise die hexadezimale Auflistung von Dateiinhalten die physische Anordnung nahezu 1:1 wiedergibt – bis auf diejenigen Strukturen, mit denen die Satz- und Blockformate verwaltet werden..

Zweidimensionale Strukturen, wie es die Key-Value-Paare sind, können problemlos in Tabellenform dargestellt werden. Es wird keine Sortiertheit oder irgendeine andere Ordnung erwartet. Es ist auch nicht unbedingt Eindeutigkeit erforderlich: In TS2 können dieselben Keys mehrfach auftreten, aber in TS1 und TS3 nicht, was aber keine Eigenschaft der Tabellen, sondern aufgabenspezifisch ist. Schlüsseleindeutigkeit kann von Fall zu Fall beim Anlegen der Tabellen vorgegeben werden, muss es aber nicht. Eindeutigkeitsprüfungen erfordern zusätzlichen Aufwand, der auch extern, also in der die Daten erzeugenden oder in der sie lesenden Applikation, erbracht werden kann.

Tabellen sind neutrale, zweidimensionale Gefäße für beliebige Informationselemente, vorausgesetzt, diese enthalten keine offengelegten Wiederholstrukturen. Es sind auch Tabellen denkbar, in denen nicht alle Spaltenwerte einer Zeile vorhanden sind, zum Beispiel metereologische Bebachtungstabellen, in denen Temperatur-, Feuchte- oder Druckwerte fehlen können, weil dafür keine Daten vorliegen. Spalten, die solche Fehlwerte erlauben, heissen NULL-fähig. Im Kontext der hier vorgelegten Mischaufgaben wird auch eine solche Spalte auftreten.

Die NULL-Fähigkeit dient aber nicht nur dazu, das Fehlen einer Information zu signalisieren, sondern sie erlaubt eine verbesserte Semantik. So bedeutet zum Beispiel das Datum 9999-12-31 in sequentiellen Beständen im allgemeinen, dass die damit gemeinten Daten beliebig lange gültig sind, oder dass ein dort erwarteter Datumswert nicht existiert. Der Wert NULL leistet denselben Dienst, ohne ein fiktionales Datum verwenden zu müssen, das programmiertechnisch „erkannt“ und gesondert verarbeitet werden muss. Ein weiteres Beispiel: Bei der Bestandsführung in alten Systemen kam es durchaus vor, dass negative Anzahlen verfügbarer Objekte in Wirklichkeit bedeuteten, dass das Lager zwar diesbezüglich leer war, aber bereits ein Bedarf für die genannte Anzahl dieser Objekte – ohne Vorzeichen, versteht sich – bestand. Relationale Tabellen können natürlich auch für solche Tricks mißbraucht werden, aber es ist doch offensichtlich besser, den Lagerbestand und die Reservierungen voneinander zu trennen, wobei das Fehlen von Reservierungen mittels NULL ausgedrückt werden sollte, nicht aber durch die Zahl Null.

3.3 RDBMS = Relational Database Management System(s)

… beherbergen die relationalen Tabellen und unterstützen die Arbeit mit ihnen. Das Kapitel „RDBMS: Grundlagen" im Band 5-C2 beschreibt anhand von Beispielen die RDBMS-Funktionsweise aus Anwendersicht. Daher können es wir hier kurz machen: Die Tabellendaten werden physisch zwar mit Hilfe des Dateisystems gespeichert, sind aber für Anwendungsprogramme, also von „außen", nur durch das jeweilige RDBMS hindurch zugreifbar. Zur Kommunikation mit dem RDBMS dient eine spezielle Sprache, die SQL genannt wird. Für das Programmieren ist meistens nur deren Bestandteil namens DML = Data Manipulation Language erforderlich. Die DDL = Data Definition Language ermöglicht es, dem jeweiligen RDBMS mitzuteilen, welche Strukturen es errichten, ändern oder löschen soll. Per DCL = Data Control Language werden Zugriffsrechte vergeben oder entzogen. Die SQL besteht aus diesen drei Teilen.

Ohne näher auf die Details einzugehen, soll hier festgehalten werden, dass DDL und DCL gegenüber den file-system-basierten Datenhaltungen eine erhebliche Innovation darstellen, denn Tabellen und andere Objekte in einem RDBMS sind wegen der DDL technisch selbstbeschreibend, und die differenzierte Vergabe von Zugriffsrechten per DCL bis hinunter auf die Ebene einzelner Tabellenspalten eröffnet Geheimhaltungsmöglichkeiten nach dem „need to know"-Prinzip.

Alle RDBMS implementieren die zentralen Eigenschaften, also die Kapselung der Tabellen und die Zugriffe per DML, sowie die Strukturvorgaben per DDL und – außer bei rudimentären RDBMS, die beispielsweise für geräteinterne Anwendungen eingesetzt werden – die Zugriffsregulierung per DCL. Teile der DML wurden ANSI-standardisiert, aber jedes RDBMS weist DML-Besonderheiten auf. Bei DDL und DCL gibt es Ähnlichkeiten, aber auch große Unterschiede zwischen den RDBMS. Selbstverständlich unterscheiden sich die Leistungsspektren der RDBMS von den Single-User-Maschinen über Server für multiple (aber nicht allzuviele) Benutzerprozesse bis hin zu Großrechnern gravierend.

Das Band-5-C2-Kapitel „RDBMS: Programmierung" beschreibt detailliert, welche Zugriffsmöglichkeiten den Programmen zur Verfügung stehen und wie sie eingesetzt werden können. Deshalb werden im folgenden Text nur diejenigen Aspekte herausgearbeitet, die für das aktuelle Thema, also das Mischen unsortierter Bestände, wichtig sind.

3.4 Tabellen-Direktzugriffe

… sind für RDBMS nur Spezialfälle, denn diese Systeme arbeiten mengenorientiert: Die DML operiert beim Lesen per SELECT, Query genannt, im Prinzip auf Matrizen, die aus den Zeilen und Spalten von Tabellen bestehen, wobei die Spalten mehrerer Tabellen mit JOIN zusammengeführt werden können. Mehr oder minder raffinierte Auswahlkriterien, Prädikate genannt, die ihrerseits tabellengesteuert operieren können, dienen zur Filterung von Daten. Ergebnismengen können mittels UNION miteinander verschmolzen werden, wenn die Spalten der Teilmengen hinsichtlich Reihenfolge und Typ zusammenpassen. Gegenüber sequentiellen Dateien, die Satz für Satz verarbeitet werden müssen, eröffnen sich so völlig neue Möglichkeiten. Ein relational arbeitendes Programm muss nicht auf alle Datenelemente einer physischen oder virtuellen Zeile zugreifen, sondern kann sich auf die jeweils relevanten Spalten beschränken.

Die TS1-Tabelle und die TS3-Tabelle sind hinsichtlich der Key-Spalte eindeutig, die TS2-Tabelle aber nicht. Ein Direktzugriff setzt aber Eindeutigkeit voraus. Das Großrechner-RDBMS Db2® bietet eine eigene Syntax für Direktzugriffe an, die sogenannten Singleton SELECTs. Wenn ein Programm einen solchen SELECT absetzt und wider Erwarten mehrere Zeilen das Suchkriterium erfüllen, dann meldet Db2 einen Fehlercode zurück. Das sollte immer Anlass für eine Fehlerbehandlung – im allgemeinen mit nachfolgendem Abbruch – sein, denn dann stimmen die Erwartungen und die Realität nicht überein. MySQL kennt keine Singleton SELECTs. Die Verantwortung für die Eindeutigkeit eines Zugriffs liegt jedenfalls immer zuerst beim Datenbankadministrator, der die richtigen Vorgaben machen muss.

Mit und ohne Singleton SELECT muss die Programmiererin oder der Programmierer – in Kenntnis der Tabellenstruktur(en) – eine Query aufbauen, deren Prädikat exakt von einer oder von keiner Zeile erfüllt wird. Kann das RDBMS die Eindeutigkeit nicht selbst gewährleisten, muss sie applikatorisch erzwungen werden.

3.5 Indizes: Ohne Redundanz keine Performanz

Auf dem gegenwärtigen technologischen Stand ist jedes Datenbanksystem, auch ein nicht-relationales, auf Indizes angewiesen, um die gespeicherten Daten effizient zu erschließen. Indizes verwenden Bestandteile der eigentlichen Zeilendaten, gewöhnlich Schlüsselelemente, um schnelle Zugriffe zu ermöglichen, sowie um gegebenenfalls Eindeutigkeit zu erzwingen. Sie enthalten also redundante Daten. Je mehr Indizes ein bestimmter Bestand aufweist, um so höher ist der Aufwand, um sie zu führen, und um so mehr Speicher wird dafür gebraucht. Daher werden im Allgemeinen nur so viele Indizes angelegt, wie aufgrund der meistverwendeten Zugriffsformen zwingend benötigt werden.

Der einfachste Fall eines Index liegt bei einer relationalen Tabelle vor, wenn eine einzelne Spalte diejenigen Daten enthält, die 1:1 zur Zugriffsbeschleunigung dienen können. Das ist im vorliegenden Beispiel die jeweilige Key-Spalte. Im Fall von TS1 und TS3 muss der Schlüssel eindeutig sein, das heißt, im Index darf nur ein einzelner Eintrag für jeden Schlüsselwert existieren. Die Suche nach einer TS1- oder TS3-Zeile stützt sich dann zunächst auf den jeweiligen Indexeintrag, der auf die zugehörige Datenzeile verweist. Der Suchvorgang besteht also aus einer Suche im Index, die effizient ablaufen muss, gefolgt von einem Direktzugriff auf die Daten.

Im Fall TS2 sind mehrere Einträge pro Key zulässig. Einen Index über eine solche Spalte nennt man einen „non-unique index". TS2 eignet sich eindeutig nicht für Direktzugriffe. Dennoch kann ein uneindeutiger Index zur Zugriffsbeschleunigung dienen, da alle Zeilen mit den passenden Schlüsselwerten durch eine Index-Suche gefunden werden können, was häufig weit schneller als eine Suche in den Tabellendaten selbst ist.

Auch über der Value-Spalte könnte ein Index errichtet werden, aber es liegt auf der Hand, dass dies ein wenig nützlicher Index wäre. Eindeutigkeit wäre darin auf keinen Fall gewährleistet. Ein solcher Index könnte die Filterung auf bestimmte Werte in der Value-Spalte unterstützen, aber schlimmstenfalls enthielte er bei identischen Inhalten der Value-Spalte in allen Zeilen nur einen einzigen Eintrag mit zahlreichen Verweisen auf die ebenso zahlreichen Tabellenzeilen. Solch einen unsinnigen Index würde im vorliegenden Fall wohl kaum jemand anlegen, aber in der Praxis wurden mehrfach Indizes angetroffen, die auf Spalten mit einer extrem kleinen Wertestreuung basieren. Nicht-eindeutige Indizes wollen also wohl erwogen sein, weil ihre Verwaltung schlimmstenfalls der Performance mehr schadet als nutzt.

Indizes bestehen häufig aus mehreren Spalten. Da Indizes eigentlich mehr ein Aspekt der Tabellenphysik als der Fachlichkeit sind, spielen bei ihrem Aufbau technische Überlegungen die Hauptrolle. Häufig, aber nicht immer, hilft die ERM beim Ermitteln des Indexaufbaus, weil Existenzabhängigkeiten von Zeilen damit einhergehen, dass einem gegebenen Primärschlüssel mindestens ein weiteres Schlüsselelement hinzugefügt wird. Der neue Schlüssel wird also länger, was bei tiefgestaffelten Hierarchien zu enorm langen Schlüsseln führen kann. Solchermaßen abgeleitete Indizes sind aufgebläht, also langsam und inffektiv. Man kann sich in diesen Fällen mit sogenannten Kunstschlüsseln behelfen, opfert dann aber meistens die Zugriffsvorteile, die bei der Verarbeitung entlang „natürlicher" Schlüsselfolgen eintreten.

Die Definition schlanker Indizes unter Beibehaltung der fachlich definierten Schlüsselfolgen erfordert also ein gutes Wissen über die Zugriffsnotwendigkeiten einerseits und über den optimalen Schlüsselaufbau andererseits. Wenn Eindeutigkeit nicht per Index erzwungen werden muss, kann auch der Verzicht auf Indizes eine Optimierungsmaßnahme darstellen. RDBMS wie MySQL und Db2 bevorzugen das Suchen im Hauptspeicher, wenn Indizes schlecht aufgebaut sind, oder wenn ihre Nutzung die Zugriffskosten erhöhen würde.

3.6 Hash-Indizes und B-Baum-Indizes

… sind wohl die häufigsten Implementierungen für Indizes in den RDBMS. Sie sind grundverschieden und haben jeweils eine andere Historie. B-Baum-Indizes blicken auf eine lange Tradition in schlüsselsequentiell organisierten Beständen zurück, während Hash-Indizes schon früh für schnelle Direktzugriffe eingesetzt wurden. Beispielsweise bot – und bietet – das IBM-Datenbanksystem IMS DB mit der Organisationsform HDAM („hierarchical direct access method") eine hash-basierte Lösung und mit HIDAM („hierarchical indexed direct access method") eine B-Baum-Index-Lösung an. Beide haben ihre Meriten und erfordern daher eine sorgfältige Abwägung bei der Entscheidung für HDAM einerseits oder HIDAM andererseits.

3.6.1 Hash-Indizes

In Hash-Indizes wird die Plazierung der Indexelemente zunächst durch Berechnung ihrer gewünschten Position anhand von numerischen Schlüsseleigenschaften vorgenommen: Jeder Schlüssel kann als eine Bitfolge interpretiert werden, die einer Zahl äquivalent ist. Der Indexbereich einer Hash-Tabelle (Index Area) ist in einen Primärbereich (Hash Index) und einen Overflow-Bereich (Overflow Index) aufgeteilt, die im allgemeinen verschieden groß sind. Bei gegebener Größe des Primärbereichs und Schlüssellänge wird errechnet, wieviele Indexeinträge in den Primärbereich passen.

Beim Einfügen einer Zeile wird für deren Schlüssel die Eintragsposition beispielsweise durch eine Modulo-Berechnung ermittelt, wobei der Schlüssel im Zähler und die Anzahl n möglicher Indexeinträge im Nenner stehen. Der Teilungsrest ist dann die primäre Zielposition von 0 bis n-1. Dieses oder ein damit verwandtes Vorgehen nennt man „hashing" – unabhängig davon, wie die Berechnung tatsächlich stattfindet. Ist der Index nicht leer, kann es beim Einfügen zu Kollisionen mit bereits bestehenden Indexeinträgen kommen, welche denselben Hash-Wert besitzen. Das ist bei Zeilen mit identischen Schlüsseln unvermeidbar, wenn keine Schlüsseleindeutigkeit gefordert ist. Zeilen mit anderen Schlüsseln können jedoch aufgrund des Hashing-Verfahrens dieselben Zielpositionen wie bestehende Zeilen zugewiesen bekommen. Solche Schlüssel nennt man Synonyme. In diesen beiden Fällen werden die neuen Indexeinträge sequentiell im Overflow-Bereich abgelegt, wohin die Primäreinträge verzeigern. Bei nachfolgenden Kollisionen können Verweisketten auch im Overflow Index entstehen. Jeder Indexeintrag zeigt somit sowohl auf die zugehörige Tabellenzeile als auch gegebenenfalls auf einen weiteren Indexeintrag, der wegen eines identischen Schlüssels oder eines Synonyms im Overflow Index liegt.

- Diese Datenorganisation ist hauptsächlich für Direktzugriffe beim Suchen gedacht. Zunächst wird die primäre Zielposition aufgrund der Suchschlüssels berechnet. Falls dort ein anderer Schlüssel gespeichert ist, muss die Verkettung zum oder im Overflow-Bereich so lange durchlaufen werden, bis die gewünschte Zeile gefunden ist, oder bis die Suche ergebnislos endet. Bei einem Treffer erfolgt ein weiterer Zugriff auf die jeweilige Tabellenzeile.
- Daraus ergibt sich, dass die Größe des Primärbereichs klug gewählt werden muss. Ist sie zu klein, dann kommt es zu vielen Verweisen in den Overflow-Bereich. Ist sie zu groß, wird Platz verschwendet.
- Eindeutigkeit kann, muss aber nicht erzwungen werden. Soll ein Hash Index eindeutig sein, dann wird jede neue Zeile daraufhin geprüft, ob für ihren Schlüssel bereits ein Indexeintrag existiert. Ist das der Fall, wird die neue Zeile abgelehnt.
- Beim sequentiellen Lesen aus einer Hash-Tabelle ist ohne ORDER BY keine bestimmte Reihenfolge zu erwarten. Das RDBMS wird die Indexeinträge vermutlich selbst dann ignorieren, wenn die jeweilige Query Kriterien für Schlüsselspalten nennt. In diesem Fall findet das Lesen und die Filterung ausschließlich auf der Ebene der Tabellendaten in deren physischer Abfolge statt.

Die folgende Abbildung visualisiert schematisch die Datenorganisation mit Hash Index, wobei nur wenige Hash-Einträge und deren Sekundärverweise in den Overflow Index, sowie ein einziger verketteter Overflow-Eintrag gezeigt werden. Die eigentlichen Zeilendaten (Rows) werden anfangs in der Table Area in Zugangsfolge abgelegt, können aber später nach Löschungen und erneutem Belegen der dadurch entstandenen Lücken durchaus in chaotischer Reihenfolge stehen.

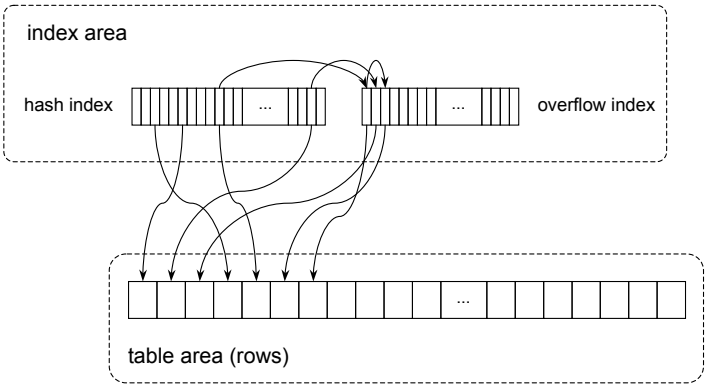


Abbildung 3.1: *Hash Index, Overflow Index und Verweisstruktur*

Es liegt auf der Hand, dass durch Löschungen einzelner Zeilen verstreute Lücken in der Table Area entstehen, die eine Freiplatzverwaltung nötig machen. Diese Löschungen führen zugleich dazu, dass Einträge im Hash Index oder im Overflow Index ungültig werden. Das erfordert lokale Reorganisationen der betroffenen Verweise und eventuell eine Freiplatzverwaltung des Overflow Index.

Unter dem Strich ermöglicht das Hash-Verfahren sehr effiziente Zugriffe bei Tabellen, die hauptsächlich per Direktzugriff bearbeitet werden. Dementsprechend bieten etliche RDBMS es an. MySQL erlaubt Hash Indexes allerdings nur für MEMORY-Tabellen, also Tabellen, die im Hauptspeicher geführt werden. Daher scheidet diese Option im vorliegenden Anwendungsfall aus, aber sie kann in anderen Fällen durchaus nützlich sein. In Db2 wurden Hash-Indizes ab 2012 eingeführt, 2022 aber bereits wieder zum Auslaufmodell erklärt. Das ist möglicherweise dadurch begründet, dass IBM zunehmend die Flash-Technologie für Hintergrundspeicher favorisiert. Dadurch entfallen die Restriktionen rotierender Medien und somit auch die Hash-Zugriffsvorteile – zumindest teilweise.

3.6.2 B-Baum-Indizes

Balanced Tree Indexes oder B-Baum-Indizes bestehen aus hierarchisch gestaffelten Indexblöcken. Wenn es zwei oder mehr Indexebenen gibt, verweist der Top-Block auf Blöcke der zweiten Ebene, welche wiederum auf Blöcke der dritten Ebene verweisen etcetera. Die Blöcke der untersten Ebene enthalten die Zeiger auf die Tabellendaten. Deren Zeilen werden nicht einzeln adressiert, sondern sind zu Blöcken zusammengefasst. Die „Abbildung 3.2: Primärindex-Baum und Verweisstruktur“ auf Seite 25 zeigt den prinzipiellen Aufbau einer Tabelle mit B-Baum-Index.

Diese Baumstruktur heißt balanciert, weil ein einheitlicher Indexaufbau mit möglichst wenigen Ebenen angestrebt wird. Das wird dadurch erreicht, dass ein voller Indexblock beim Zugang eines weiteren Indexeintrags in zwei etwa gleich große Indexblöcke gesplittet wird, sofern es nicht möglich ist, einen neuen Indexblock – zum Beispiel im Fall stetig steigender Indexwerte – auf derselben Ebene anzufügen. Ein Split oder ein neuer Indexblock bewirken einen neuen Eintrag in der nächsthöheren Indexebene etcetera, also gegebenenfalls bis hinauf zum Top-Indexblock.

Wenn dieser voll ist und ein neuer Top-Indexeintrag hinzukommt, wird der bisherige Top-Block entweder gesplittet (beim Einfügen „zwischendrin") oder es wird ihm ein weiterer Block zur Seite gestellt, der den neuen Eintrag aufnimmt (beim aufsteigenden Einfügen). Zusätzlich wird ein neuer Top-Block angelegt, der auf die beiden Blöcke eine Ebene tiefer verzeigert. Der Indexbaum wächst also immer höher.

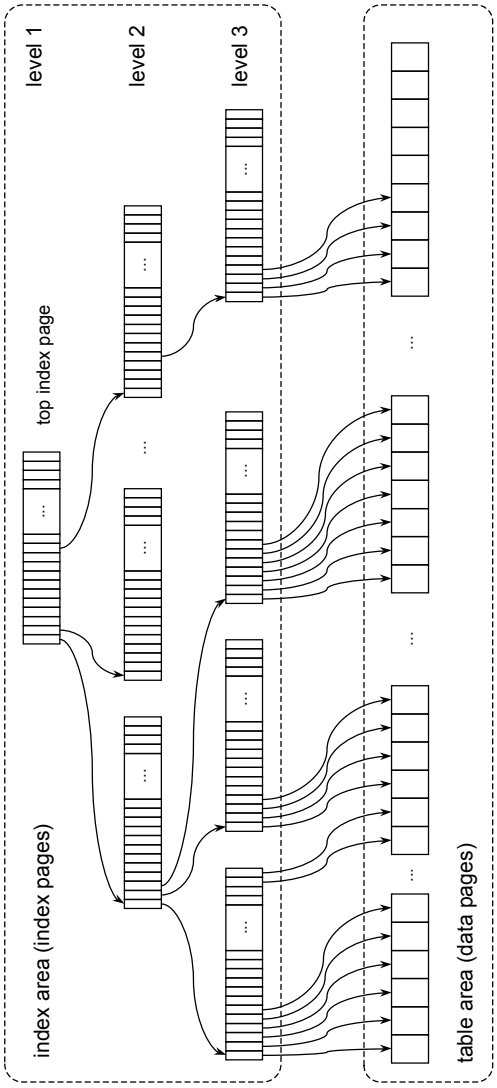


Abbildung 3.2: *Primärindex-Baum und Verweisstruktur*

Anmerkungen

- Das RDBMS nutzt den Primärindex – bei Db2: Cluster Index – hauptsächlich dafür, die Plazierung neuer Zeilen zu optimieren. Bei gegebenem Indexaufbau determiniert jeder neu eintreffende Zeilenschlüssel, welcher Top-Indexeintrag zu verwenden ist, um den richtigen Indexblock in der zweiten Ebene anzusteuern. Dort und auf gegebenenfalls weiteren Indexebenen wiederholt sich der Vorgang, bis schließlich derjenige Table Area Block erreicht ist, in den die neue Zeile hinsichtlich ihres Schlüssels passt. Dort wird sie eingefügt, sofern dieser Block noch Reserveplatz aufweist. Ist das nicht (mehr) der Fall, dann wird der

Block gesplittet und für den neuen Block ein neuer Indexeintrag erzeugt. Einfügungen von Zeilen mit stetig wachsenden Schlüsseln – aus sortierten Daten oder beispielsweise aus Timestamps – bewirken, dass die neuen Blöcke in der Table Area kontinuierlich aufeinanderfolgen und der Indexbaum nach rechts (sowie später nach oben) wächst. Der Indexbaum bleibt dadurch balanciert.

- Blöcke, die aus Splits stammen, werden in bislang unbelegten Zonen der jeweiligen Teilstruktur gespeichert. Das bewirkt, dass die Blockzeiger der jeweils nächsthöheren Ebene (bei den Tabellendaten also die Indexverweise der untersten Ebene) nach und nach „verwildern", also nicht kontinuierlich aufsteigend angeordnet sind. Dies erfordert bei sequentiellen Verarbeitungen zusätzliche Zugriffe auf eventuell weit entfernte Blöcke, was bei rotierenden Medien mit erheblichen Latenzzeiten einhergeht.
- Das Verfahren reagiert also empfindlich auf gestreute Einfügungen in eine bereits voll belegte Tabellen- und Indexstruktur. Dann kommt es zu zahlreichen Splits in beiden Strukturen und folglich zu einem raschen Indexwachstum, obwohl das möglicherweise nicht nötig wäre. Beim Reorganisieren von Tabellen, für welche mit gestreuten Einfügungen zu rechnen ist, muss daher genügend viel Reserveplatz in beiden Strukturen vorgegeben werden.
- Das Löschen von Tabellenzeilen führt wegen der Zusammenfassung zu Blöcken zunächst nicht zur block-externen Platzfreigabe. Erst wenn die Zeilen eines ganzen Blocks gelöscht sind, kann dessen Platz wiederverwendet werden. Ausserdem entfällt dann der zugehörige Indexeintrag. Nach Löschung vieler Datenblöcke kann es daher zur Löschung eines unteren Indexblocks kommen, sowie von dessen Indexeintrag auf der nächsthöheren Ebene etcetera. Die Freiplatzstrategie des RDBMS entscheidet darüber, was mit diesen Lücken geschieht. Bei einem bekannten indexsequentiell organisierten Dateisystem führte die fehlende Freiplatznutzung dazu, dass die betroffenen Dateien über die Platte „vorwärts krochen", also immer größer wurden, obwohl sie nicht mehr aktive Daten speicherten.
- B-Baum-Indizes erzwingen keine Eindeutigkeit. Zeilen mit identischem Schlüssel wie bereits existierende Zeilen werden möglichst in der Nähe ihrer Vorgänger eingefügt. Alternative Indizes sind beim B-Baum-Verfahren stets desorganisiert, weil die Indexverweise auf die Daten nicht in der Folge des führenden Index angeordnet sein können. Wenn Eindeutigkeit gefordert ist, muss das RDBMS bis auf die Zeilenebene vorstoßen, um doppelte Schlüssel zu verhindern.
- Direktzugriffe per Singleton SELECT beginnen mit dem Top-Indexblock, wo der passende Indexeintrag gesucht wird, durchlaufen dann in analoger Weise alle Indexebenen entlang der Verweise und münden schließlich in einem Datenblock, der die gesuchte Zeile enthalten kann. Wenn die Zeile fehlt, wird ein Fehlercode rückgemeldet. In RDBMS wie MySQL, die keinen Singleton SELECT kennen, besteht das Zugriffsergebnis immer aus einer Zeilenmenge oder der leeren Menge. Erstere darf stets nur eine einzige Zeile enthalten, was durch das Zusammenwirken eines eindeutigen Index mit der Vorgabe aller darin enthaltenen Spalten im Equal-Prädikat der Query erzwungen werden muss.
- Beim sequentiellen Lesen aus einer b-baum-organisierten Tabelle ist ohne ORDER BY keine bestimmte Reihenfolge zu erwarten. Das RDBMS liefert die herausgefilterten Zeilen in ihrer physischen Abfolge, was manchmal nützlich sein kann, häufig aber nicht gewünscht ist. Je nach RDBMS und nach dem Zustand des betreffenden Index kann die Vorgabe von ORDER BY zur Folge haben, dass auf die Zeilen der jeweiligen Basistabelle direkt in der gewünschten Sortierfolge zugegriffen wird, besonders nach Reorganisationsläufen. Andernfalls müssen die ausgefilterten Tabellenzeilen im RDBMS sortiert werden. Im Db2 kann beim Lesen mit ORDER BY keine Zugriffssperre für konkurrierende Zugreifer auf die aktuell in Bearbeitung befindliche Zeilenposition gesetzt werden. Andere RDBMS halten es vermutlich genauso.

B-Baum-Indizes haben sich vor allem für die Organisation von Beständen bewährt, die sich an klassischen Primärschlüsseln orientieren. Der Primary Index – bei Db2: der Cluster Index – bietet auch im Online-Betrieb bei unvorhersagbaren Zugriffsmustern eine effiziente Speicherungsstruktur, die allerdings mittels regelmäßiger Reorganisationen in gutem Zustand gehalten werden muss.

Bei Sekundärindizes auf B-Baum-Basis will dagegen wohl erwogen sein, ob sie erstens überhaupt nötig sind und ob sie zweitens eine hinreichende Wertestreuung aufweisen. Letztere kann durch Aufnahme zusätzlicher – identifizierender – Spalten in solche Indizes erreicht werden, allerdings um den Preis höheren Platzbedarfs.

3.7 INSERT, SELECT, UPDATE und DELETE

… sind die Grundvokabeln in der Kommunikation von Anwendungsprogrammen mit den RDBMS. In Jahrzehnten der RDBMS-Weiterentwicklung wurden immer raffiniertere Varianten dieser DML-Verben entwickelt, die es vor allem Endbenutzern erlauben, ganz besonders ausgefeilte Operationen auf relationalen Beständen ablaufen zu lassen. Programmierer(innen) sollten dagegen der Versuchung widerstehen, sich solcher Techniken zu bedienen, wenn es nicht wirklich unumgänglich ist. Andernfalls müssen sie sich womöglich mit üblen Performanceproblemen herumschlagen.

Das überrascht und ernüchtert viele, die in Kursen gelernt haben, was für fantastische Möglichkeiten in den heutigen RDBMS stecken. Dort wird immer noch das Paradigma gepredigt, dass es in der schönen neuen RDBMS-Welt völlig ausreicht, das WAS per DML zu formulieren und das WIE dem Datenbanksystem zu überlassen. Schon das vorangehende Unterkapitel über Indizes wäre völlig überflüssig, wollte man dieser verführerischen Parole wirklich Glauben schenken. Es hat Ihnen, geschätzte Leserinnen und Leser, aber hoffentlich vermittelt, dass eine intime Kenntnis der Datenbankstrukturen sowohl vor explodierendem Platzbedarf als auch vor hohen Zugriffszeiten zu schützen vermag.

Es gibt aber noch weitere Gründe, warum die DML-Programmierung sich möglichst einfacher Mittel bedienen sollte. Einer davon ist Portabilität: Je standardkonformer die verwendete DML ist, um so eher können die betreffenden Programme auf andere Plattformen portiert werden. Ein weiterer, wesentlich wichtigerer Grund ist der, dass die Kontrolle über die Zugriffe stets beim Programm liegen sollte. Automatiken des RDBMS, mit denen beispielsweise große Datenmengen kopiert oder manipuliert werden, sind zu meiden. Sie verursachen hohe Zugriffskosten, blockieren Bestände für konkurrierende Zugriffe, und sie können wegen zu großer Mengen nach langer Laufzeit scheitern, womit erhebliche Aufwendungen sinnlos verpuffen.

Das Band-5-C2-Kapitel „RDBMS: Programmierung“ geht, auch anhand vieler Beispiele, sehr detailliert auf die Programmierung mittels DML ein. Daher wird im folgenden nur das Rüstzeug für die aktuelle Aufgabe vermittelt. Dabei steht im Vordergrund, dass Zugriffe auf relationale Tabellen nicht nur einen Austausch der Zugriffs-Basismaschine bedeuten, sondern auch ein anderes Vorgehen bezüglich der Realisierungsoptionen ermöglichen. Was aus der Sicht sequentieller Verfahren als evident, ja trivial gilt, kann relational schwierig sein, auch weil die RDBMS völlig neue Möglichkeiten und Gegebenheiten mit sich bringen. Andererseits ist ja gerade das im vorhergehenden Kapitel behandelte Problem sequentiell nicht mit vertretbarem Aufwand lösbar. Direktzugriffstechniken, insbesondere solche auf relationaler Basis, erlauben es uns, die Aufgabe und ihre möglichen Lösungen aus einer neuen Sicht zu betrachten und eine gute Lösung zu finden.

3.8 Mischen ungeordneter Bestände auf relationaler Basis

Relational arbeitende Programme sind Kunden = Clients eines RDBMS, das sich entweder als Server verhält, oder das in die Anwendung integriert ist. MySQL und viele andere – vor allem die großen – RDBMS agieren als Server, welche die relationalen Bestände mit Hilfe des File System vor direkten Zugriffen aus der Aussenwelt schützen. Bildlich kann man sich das so vorstellen, wie es die „Abbildung 3.3: Programme im RDBMS-Umfeld“ auf Seite 28 zeigt.

In den späteren Kapiteln wird die im folgenden erarbeitete – relativ primitive – Problemlösung durch zunehmend leistungsfähigere und elegantere Alternativen ersetzt. Betrachten Sie, geschätzte Leserinnen und Leser, jedoch den nun beginnenden Weg als das eigentliche Ziel. Auch das Programm SQL_Example3, das im Kapitel „6 Mischen mit Sortieren und Verdichten“ ab dem Unterkapitel „6.3 Dritte relationale Misch-Alternative“

auf Seite 142 ff entworfen wird, ist nur ein Hilfsmittel. Es geht hier *nicht* darum, die vielleicht beste Lösung für ein – Ihnen möglicherweise als willkürlich gewählt erscheinendes – Programmierproblem zu präsentieren, sondern das Potential beim Entwerfen einer Mischverarbeitung im RDBMS-Umfeld soll demonstriert werden, um *Ihre* Fantasie anzuregen.

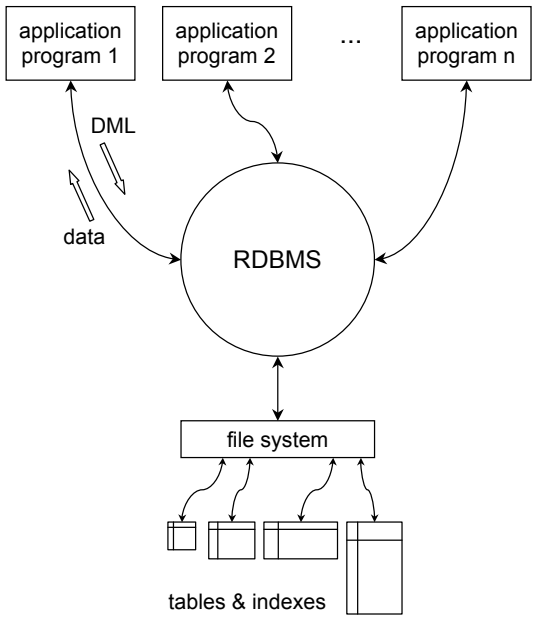


Abbildung 3.3: *Programme im RDBMS-Umfeld*

Anmerkungen

- Programme können mit ihrem RDBMS auch per DDL oder DCL kommunizieren. Das ist für die Anwendungsprogrammierung aber unüblich und wurde deshalb in der obigen Grafik weggelassen.
- Im aktuellen Fall wird die Konkurrenzverarbeitung weitgehend ausgeklammert, welche das jeweilige RDBMS dadurch ermöglichen kann, dass es quasiparallele Zugriffe auf dieselben Tabellen durch mehrere Programme zulässt. Dadurch vereinfacht sich im vorliegenden Beispiel das Bild erheblich, wie das folgende Datenflussdiagramm zeigt.

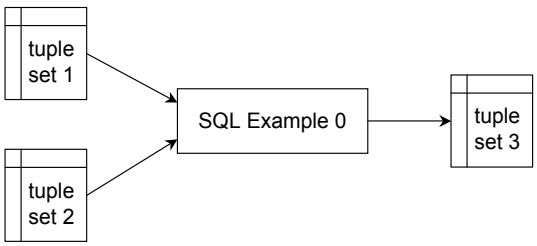


Abbildung 3.4: *Relationale Lösung der Mischaufgabe*

Diese Grafik verändert die Sicht auf die vorliegende Aufgabe grundlegend:

- Die zu verarbeitenden Datenmengen TS1 und TS2 befinden sich in relationalen Tabellen, wobei offen ist, wie sie dahin gekommen sind. Anders als bei sequentiellen Dateien, die mit einem Editor bearbeitet werden können, wird ein Ladeprogramm oder mindestens eine DML-Anweisung benötigt, um sie zu speichern.
- Das Programm „SQL Example 0" muss gegenüber dem RDBMS – hier: MySQL – die zu bearbeitenden Tabellen identifizieren. Es genügt nicht, einfach die beiden Eingaben und die Ausgabe zu öffnen, denn diese sind nicht anonym. Ausserdem muss das Programm die erforderlichen Zugriffsberechtigungen für die Tabellen besitzen, also für TS1 und TS2 leseberechtigt, sowie für TS3 schreibberechtigt sein.
- Sequentielle Lesezugriffe auf TS1 und TS2 erfolgen dadurch, dass per SELECT die Ergebnismenge identifiziert wird und das RDBMS anschließend die Ergebniszeilen auf Anforderung übergibt. Beim Db2 erfolgt diese Anforderung unter COBOL und anderen algorithmischen Sprachen mit der FETCH-Anweisung; andere Sprachen und andere RDBMS haben eigene Varianten für den Zeilenabruf.
- Die Werteaggregation in TS3 wird für jedes TS2-Tupel zunächst per UPDATE vorgenommen. Existiert die zu ändernde Zeile nicht, wird sie per INSERT neu eingefügt. Dies ist im vorliegenden Fall die entscheidende, durch Direktzugriffe ermöglichte Innovation. Der Krampf, mit multiplen Scans eventuell korrespondierende Tupel zu finden, entfällt komplett.
- Das Ergebnis ist der mit korrekten Daten gefüllte Tuple Set 3 in Form der TS3-Tabelle. Das passt allerdings nicht zu der sequentiell orientierten Fortschreibungslogik, bei welcher der Tuple Set 3 beim nächsten Lauf die Rolle des Tuple Set 1 einfach dadurch übernehmen kann, dass er dem Programm per Parameter auf der TS1-Position angeboten wird. An die Stelle der jetzigen TS2-Daten träte bei einer traditionellen Folgeverarbeitung ein neuer Tuple Set 2.
- Um diesen Rollentausch dennoch möglich zu machen, müsste der bisherige TS1-Inhalt gelöscht, dann der TS3-Inhalt nach TS1 kopiert und anschliessend aus TS3 gelöscht werden. Ausserdem wäre der Inhalt von TS2 auszutauschen. Das ist alles derart umständlich, dass man sich etwas Neues überlegen muss. Genau das passierte, als die ersten Datenbanksysteme eingeführt wurden.
- Alternativ zur vorhergehenden Abbildung ist daher vorstellbar, TS1 zu einem Bestand zu ernennen, der durch TS2-Tupel aktualisiert wird. TS2 hat dann die Rolle einer Delta-Tabelle, die während oder nach der Verarbeitung rückgesetzt und anschließend mit neuen Tupeln gefüllt wird. Dieses Verhalten wurde vielfach implementiert. Es ist für Back End-Systeme typisch, die von Zeit zu Zeit mit kumulierten Änderungsdaten versorgt werden.
- Im vorliegenden Fall soll jedoch nur gezeigt werden, wie die Fortschreibungslogik auf der Basis relationaler Tabellen implementiert werden kann. Der Verbleib der Bestände nach beendeter Verarbeitung wird in der eingangs geschilderten Mischaufgabe nicht thematisiert und ist für sie auch bedeutungslos. Daher hält die im folgenden skizzierte Lösung an den Rollen der drei Tuple Sets fest.

3.8.1 DDL für die Tuple Sets

Um überhaupt mit Tabellen in MySQL arbeiten zu können, müssen diese zunächst kreiert und dann mit Nutzdaten gefüllt werden. Hierfür ist es nötig, zuerst eine themenspezifische Database anzulegen, falls noch nicht vorhanden. Das neue Java-Misch-Programm SQL_Example0 soll in NetBeans® angelegt und von dort aus gestartet werden. NetBeans verwendet intern als Standard die Zeichencodierung UTF-8, aber die Tuple Sets bestehen aus Großbuchstaben ohne Umlaute und Zahlen. Daher erscheint die Codierung mit `latin1` für die neue Database als völlig ausreichend. Die folgende DDL-Anweisung in der MySQL Workbench legt diese Datenbank mit der gewünschten Zeichencodierung an, nachdem die Anmeldung bei MySQL als „root user" erfolgte.

```
CREATE DATABASE tuple_db CHARACTER SET latin1 COLLATE latin1_swedish_ci;
```

MySQL akzeptiert übrigens beliebige Groß-und Kleinschreibung in den Anweisungen mit Ausnahme von Strings, bei denen es auf die Schreibung ankommt, beispielsweise bei Passwörtern. Die COLLATE-Angabe definiert die Sortierfolge der Zeichen im gewählten Zeichensatz. `latin1_swedish_ci` ist ein MySQL-Standard. Um diese Database für alle weiteren Aktivitäten auszuwählen, ist ein USE Statement erforderlich:

```
use tuple_db;
```

Außerdem wird ein Administrator für die neue Database benötigt, da diese nicht mit den hohen Berechtigungen des „root user" geführt werden soll. Das soll stattdessen der neue Benutzer namens „msp_admin" tun. Die hierfür erforderlichen DCL-Anweisungen lauten folgendermaßen:

```
create user 'msp_admin'@'localhost' identified by '<password>';
grant all on tuple_db.* to 'msp_admin'@'localhost';
```

GRANT ALL auf alle Objekte in der Tuple_DB ist eine Art Generalschlüssel, also kein Vorbild für die differenzierte Rechtvergabe in professionellen Installationen.

Da es für die vorliegende einfache Aufgabe keinen Anlass gibt, explizit einen oder mehrere Tablespaces in der neuen Database zu kreieren, können die drei Tabellen sofort mit ihren Indizes zusammen definiert werden. MySQL legt dann implizit die zugehörigen Tablespaces an. Letztere können dazu verwendet werden, mehrere Tabellen zusammenzufassen, was hier aber nicht nötig ist. Die folgende DDL erzeugt die neuen Tabellen und ihre Indizes, sowie 1:1-Sichten auf die Tabellen, die VIEWs genannt werden:

```
CREATE TABLE TS1_0_Table
             (Tuple_Key           CHAR(1)      PRIMARY KEY,
              Tuple_Value         SMALLINT     NOT NULL);

CREATE VIEW      TS1_0_View
       AS SELECT Tuple_Key,
                 Tuple_Value
       FROM      TS1_0_Table;

CREATE TABLE TS2_0_Table
             (Tuple_Key           CHAR(1)      NOT NULL,
              Tuple_Value         SMALLINT     NOT NULL);

CREATE VIEW      TS2_0_View
       AS SELECT Tuple_Key,
                 Tuple_Value
       FROM      TS2_0_Table;

CREATE TABLE TS3_0_Table
             (Tuple_Key           CHAR(1)      PRIMARY KEY,
              Tuple_Value         SMALLINT     NOT NULL);
```

```
CREATE VIEW      TS3_0_View
       AS SELECT Tuple_Key,
                 Tuple_Value
       FROM      TS3_0_Table;
```


Die Namen der Tabellen – Basistabellen genannt – und der VIEWs enthalten den Infix '_0', um sie dem Programm SQL_Example0 zuzuordnen. Das ist ungewöhnlich, weil Tabellen normalerweise von mehreren Programmen bearbeitet werden, dient aber hier der Abgrenzung der Beispielsfälle voneinander. Gegenüber den Tabellen am Anfang des Kapitels „3.2 Relationale Tabellen“ auf Seite 19 fällt auf, dass die Spaltennamen nicht tabellenspezifisch sind. Das vereinfacht die Arbeit mit den Spaltennamen, verpflichtet aber zugleich zur Sorgfalt, damit es nicht versehentlich zur Verwechslung von Tabellen oder VIEWs kommt.

Der Nutzen der VIEWs, welche dieselben Spaltennamen wie die Basistabellen nennen, also sogenannte 1:1-Views sind, liegt hier in der Entkopplung von Tabelle und Programm(en). Programme, die über VIEWs zugreifen, sind von Änderungen der Basistabellen nur dann betroffen, wenn die in den VIEWs genannten Spalten verändert oder sogar gelöscht werden: Relativ häufig müssen Tabellen um zusätzliche Spalten erweitert werden. Davon sind nur diejenigen Programme betroffen, die unmittelbar auf die Basistabellen zugreifen. Diese Programme oder Module müssen gegebenenfalls erneut übersetzt werden. VIEWs haben noch weitere Aufgaben, beispielsweise für die Zugriffskontrolle, die aber hier nicht näher diskutiert werden.

Betrachtet man die MySQL-DDL, dann fällt auf, dass MySQL – im Gegensatz zu anderen RDBMS wie beispielsweise Db2 – keine eigenen Definitionen für die führenden Indizes benötigt. PRIMARY KEY genügt. CREATE INDEX dient in MySQL nur dazu, zusätzliche Indizes anzulegen. Soll eine Tabelle nachträglich einen führenden Index erhalten, oder soll dieser geändert werden, so ist in MySQL dafür ein ALTER TABLE Statement erforderlich. Db2 verlangt dagegen eigene Index-Definitionen neben der Tabellen-DDL.

3.8.2 DML für die Tuple Sets

Nun geht es darum, die TS1- und TS2-Tabellen mit Daten zu füllen. Es bietet sich an, dafür die Beispieldaten aus der „Abbildung 2.1: Mischen von Eingabe-Beständen mit Schlüsselduplikaten im Tuple Set 2“ auf Seite 9 zu verwenden. Mit MySQL können mehrere Zeilen mit einem einzigen INSERT Statement auf einmal eingetragen werden, was andere RDBMS so nicht anbieten:

```
INSERT INTO TS1_0_Table (Tuple_Key, Tuple_Value)
VALUES ('T',7),('F',2),('K',11),('A',-3),('N',17),('R',22),('C',6),('X',5);
```

Mit einem SELECT Statement wird die folgende Ausgabe in der MySQL Command Shell[1] – auch MySQL Monitor genannt – angefordert, wobei `'mysql> '` der Prompt, also die Eingabeaufforderung des Shell-Programms ist. Die nachfolgende Kopie aus dem Dialog zeigt die SELECT-Eingabe und die MySQL-Ausgabe unmittelbar nach dem Prompt. `Select * FROM <tabelle>` oder `Select * FROM <view>` bedeutet: Liste alle Zeilen und Spalten ohne Sortiervorgabe und Filterung auf. Möchte man nur einzelne Spalten auflisten oder alle Spalten explizit angegeben, muss der Stern durch eine komma-separierte Spaltenliste beziehungsweise durch einen einzelnen Spaltennamen ersetzt werden. In Programmen ist der Stern verpönt, auch wenn 1:1-Views verwendet werden, weil Programme stets exakte Vorgaben machen sollen. Andernfalls kann eine Tabellen- oder VIEW-Änderung zu unerwartetem Verhalten der davon betroffenen Programme führen.

Endnoten: siehe Seite 57

```
Select * from TS1_0_View;
+-----------+-------------+
| Tuple_Key | Tuple_Value |
+-----------+-------------+
| A         |          -3 |
| C         |           6 |
| F         |           2 |
| K         |          11 |
| N         |          17 |
| R         |          22 |
| T         |           7 |
| X         |           5 |
+-----------+-------------+
8 rows in set
```

Hier fällt sofort auf, dass die Zeilen auch ohne Vorgabe von ORDER BY alphabetisch aufsteigend nach der Spalte `Tuple_Key` sortiert sind. Das ist vermutlich auf die INSERT-Strategie von MySQL zurückzuführen, welche die neuen Zeilen am Primary Key ausgerichtet ablegt. Damit sollen aber keine falschen Erwartungen geschürt werden, denn im MySQL 8.0 Reference Manual wird ausdrücklich darauf hingewiesen, dass ohne ORDER BY keine bestimmte Reihenfolge garantiert wird. Das gilt auch für andere RDBMS wie Db2 , wenn nicht für alle.

Übrigens: Die Ausgabe der Ausführungszeit durch MySQL nach `'8 rows in set'` wurde – und wird auch künftig – weggelassen. Im aktuellen Fall war sie „Null", also nicht wirklich Null, sondern einfach zu klein für die Ausgabe in Sekundenbruchteilen.

Das Ergebnis bei der TS2-Tabelle nach Ausgabe-Anforderung per TABLE Statement unterscheidet sich vom bisherigen deutlich. `TABLE` bedeutet dasselbe wie `Select * FROM` und ist eine MySQL-Besonderheit.

```
INSERT INTO TS2_0_Table (Tuple_Key, Tuple_Value)
VALUES ('N',8),('I',4),('H',9),('I',-6),('F',1),('C',13),('F',-15);

TABLE TS2_0_View;
+-----------+-------------+
| Tuple_Key | Tuple_Value |
+-----------+-------------+
| N         |           8 |
| I         |           4 |
| H         |           9 |
| I         |          -6 |
| F         |           1 |
| C         |          13 |
| F         |         -15 |
+-----------+-------------+
7 rows in set
```

Hier ist die Zeilenfolge gleich der Einfügefolge, denn MySQL kann nicht „wissen", woran die neuen Zeilen ausgerichtet werden sollen. Glücklicherweise ist in beiden Tabellen keine bestimmte Reihenfolge gefordert. Immerhin liegt bereits eine große Abweichung von der „Abbildung 2.1: Mischen von Eingabe-Beständen mit Schlüsselduplikaten im Tuple Set 2" auf Seite 9 vor, denn die TS1-Zeilen werden von der Command Shell

und auch von der MySQL Workbench nicht in derselben Folge wie in der Grafik präsentiert. Sie werden deshalb auch voraussichtlich dem Programm SQL_Example0 wie gezeigt übergeben.

3.8.3 Modellierung der Daten- und Programmstrukturen

Die Entscheidung dafür, die TS2-Daten per Direktzugriff in die TS3-Tabelle einzuarbeiten, zerlegt das Programm und somit auch die Datenstrukturen in zwei Phasen: In der ersten Phase werden die TS1-Zeilen nach TS3 kopiert und in der zweiten Phase findet die Einarbeitung der TS2-Zeilen statt. Das folgende Struktogramm modelliert die Kopie von TS1 nach TS3, wodurch ein Zwischenzustand von TS3 geschaffen wird.

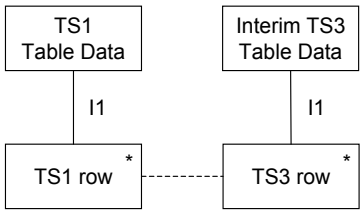


Abbildung 3.5: *Phase 1: Korrespondenz zwischen TS1 und dem TS3-Zwischenzustand*

Das sieht auf den ersten Blick so trivial aus, dass der Aufwand für die Zeichnung als überflüssig erscheint. Stünden links aber eine Gruppe von einzelnen INSERT Statements mit den unsortierten TS1-Daten und rechts die Ergebnisdaten in der Tabelle TS3, dann würde zwar die Anzahl der Statements mit der Anzahl der Zeilen übereinstimmen, aber nicht deren Reihenfolge. Das MySQL INSERT Statement mit multiplen Zeilendatenvorgaben ist ja zu einer Folge solcher Statements mit je einer Zeilenvorgabe äquivalent. Es ist nur eine Abkürzung. Hebelt das die gezeigte Korrespondenz aus? Immerhin, wenn TS1 keinen Primärschlüssel besäße und die Zeilenfolge wie in TS2 chaotisch wäre, dann träfe doch dieselbe Argumentation auch auf die erste Phase zu.

Dazu ist folgendes zu sagen:

- Der „klassische" Begriff der Korrespondenz, wie ihn Michael A. Jackson geprägt hat, bezieht sich ausdrücklich nur auf sequentielle Strukturen. Sein Kern ist die Forderung nach Synchronität, die bei sequentiellen Verarbeitungen zwingend gewahrt werden muss, soll ein aus den beteiligten Datenstrukturen abgeleitetes Programm die Ergebnisdaten in der gewünschten Folge erzeugen.
- Sobald eine aktive Instanz hinzutritt, hier also ein RDBMS, welches die angelieferten Zeilendaten nach eigenen Kriterien intern umverteilt, ist der Korrespondenzzwang scheinbar aufgehoben. Das gilt aber nur für sehr einfache Strukturen ohne Gruppenbildung. Sind die Daten gruppiert, dann muss das sie verarbeitende Programm selbstverständlich die Grenzen zwischen den Gruppen sorgfältig beachten, selbst wenn gruppenintern chaotisch verteilte, rangtiefere Schlüssel auftauchen können.
- Gegenüber den rein sequentiellen Verarbeitungen hat der Umgang mit den RDBMS den Vorteil, dass die Programme ihre Eingabedaten in einer aufgabenspezifisch sinnvollen Folge nach Wunsch anliefern lassen können. Das mag nicht effizient sein, weil das RDBMS unter Umständen hohe Aufwendungen für die Datenbereitstellung erbringen muss, aber es ist möglich. Ein Teil der Kunst guter Datenbankadministratoren und professioneller Anwendungsentwickler besteht daher darin, die Tabellen- und Indexstrukturen an die Erfordernisse der jeweils wichtigsten Massendatenverarbeitungen anzupassen.

Die zweite Phase stellt sich auf den ersten Blick als ähnlich trivial dar: TS2-Schlüssel, die in TS3 nicht vorkommen, führen zu neuen TS3-Zeilen. Die anderen münden in UPDATEs, bei denen die Tupel-Werte aggregiert werden. Es bietet sich daher an, die Korrespondenzen so zu modellieren, wie es die folgende Abbildung zeigt.

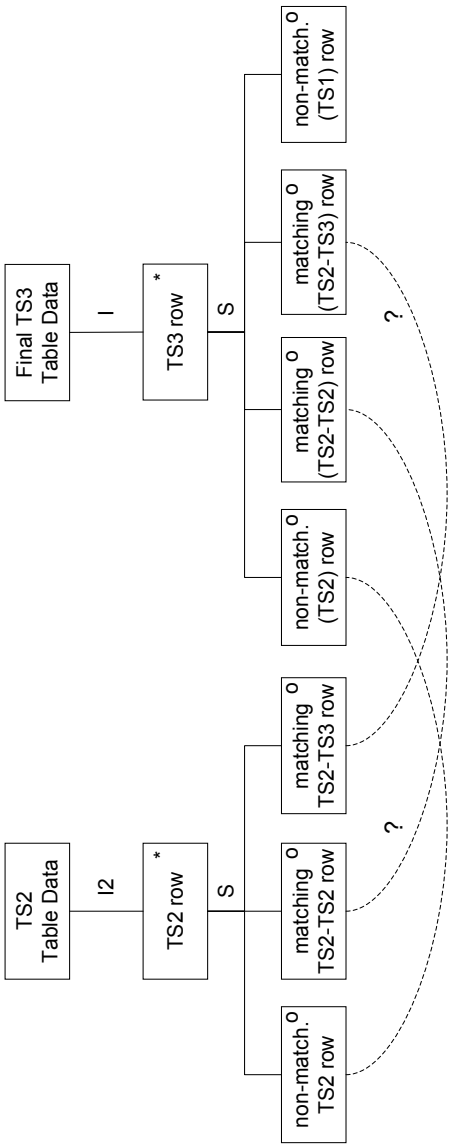


Abbildung 3.6: *Phase 2: Vermutliche Korrespondenzen zwischen TS2 und TS3*

Hinweis: Die Strukturblöcke unter der rechten S-Kontrolle nennen in Klammern die Herkunft ihrer Inhalte.

Wenn wir momentan davon absehen, dass TS2 chaotisch organisiert ist, während TS3 einen Primärschlüssel besitzt, so fällt doch sofort auf, dass mit den Anzahlen etwas nicht stimmen kann. Angenommen, in TS2 gäbe es nur Zeilen mit demselben Schlüssel wie demjenigen einer einzigen TS3-Zeile, dann wäre die Korrespondenzforderung nach gleicher Anzahl auf beiden Seiten auf jeden Fall verletzt. Die mit Fragezeichen markierten Korrespondenzen können also nicht zur Ableitung einer Programmstruktur genutzt werden.

Ganz rechts unten im Diagramm werden übrigens aus Vollständigkeitsgründen diejenigen TS3-Zeilen gezeigt, die aus TS1 stammen und in der zweiten Phase nicht geändert werden.

Dieser erste Ansatz, die Datenstrukturen der zweiten Phase und ihre Korrespondenzen zu modellieren, ist folglich gescheitert. Woran liegt das? Der Versuch, den finalen Aufbau von TS3 so zu modellieren, wie es bislang nahezu immer üblich war, also als Endprodukt, führt offensichtlich in ein Dilemma. „Nahezu immer" heisst, dass es einen Fall gab, in dem ein solcher Versuch erst gar nicht stattfand, nämlich die Entwicklung der Link-Listen-Verarbeitung im Band 1-A2. Dort lag es von Anfang an auf der Hand, dass es keine „klassische" Korrespondenz zwischen den Schulsport-Ergebnisdaten und den Ranking-Siegern geben konnte. Stattdessen führte eine Betrachtung der einzelnen Verarbeitungsschritte zum Erfolg. Dieser Ansatz lässt völlig ausser Acht, dass die beteiligten Datenstrukturen Ausschnitte aus Struktogrammen sind, die auf den oberen Rangebenen keinerlei Korrespondenzen aufweisen.

Das führt uns zur Lösung des vorliegenden Korrespondenzkonflikts:

Die Forderung nach Synchronität, die sich in der Suche nach Korrespondenzen zwischen Strukturbäumen ausdrückt, kann auch durch Schritt-für-Schritt-Korrespondenzen erfüllt werden. Wenn jeder Schritt einen bestehenden Zustand korrekt in einen neuen Zustand transformiert, muss die Gesamtheit dieser Schritte ein im Sinn der Methode korrektes Ergebnis erzeugen.

Die folgende Grafik demonstriert die Schritt-für-Schritt-Korrespondenzen im vorliegenden Fall:

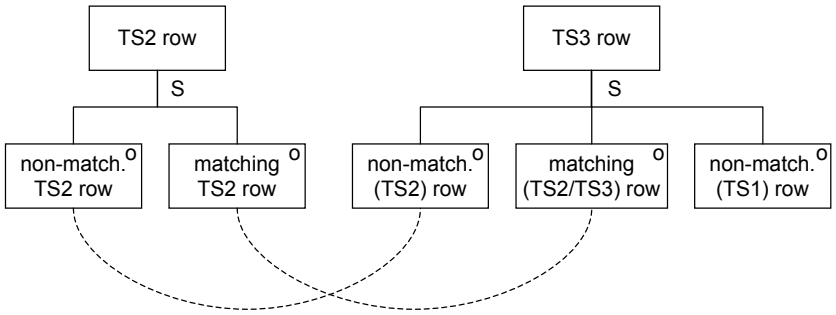


Abbildung 3.7: „*Interaktion" von TS2 mit TS3 bei übereinstimmenden oder disjunkten Schlüsseln*

Die Korrespondenzen bedeuten, dass bei jedem einzelnen Transformationsschritt entschieden wird, um welchen der Fälle es sich handelt. Im Verlauf aufeinanderfolgender Schritte kann eine neu aus TS2 (per INSERT) nach TS3 übertragene Zeile ihrerseits einen „Match" mit einer weiteren TS2-Zeile haben (Ergebnis: UPDATE). TS2-Zeilen, die von vornherein mit vorhandenen TS3-Zeilen korrespondieren, führen dagegen ausschließlich zu UPDATEs in TS3. Das wird durch die Herkunftsbezeichnung (TS2/TS3) symbolisiert.

Der aus Vollständigkeitsgründen aufgeführte Strukturblock „non-match(ing) (TS1) row" trägt nichts zur Verarbeitung in der zweiten Phase bei und kann daher bei der Bildung der Programmstruktur weggelassen werden. Somit gilt für beide Seiten dieselbe S-Kontrolle. Es erübrigt sich daher, das Ergebnis der Verschmelzung der Strukturblöcke zur Programmstruktur grafisch darzustellen, weil das strukturell wie die „TS2 row" aussähe. Da in der zweiten Phase so viele Transformationsschritte erforderlich sind, wie es TS2-Zeilen gibt, gilt dafür die I2-Kontrolle. Somit kann die Programmstruktur wie folgt abgeleitet werden.

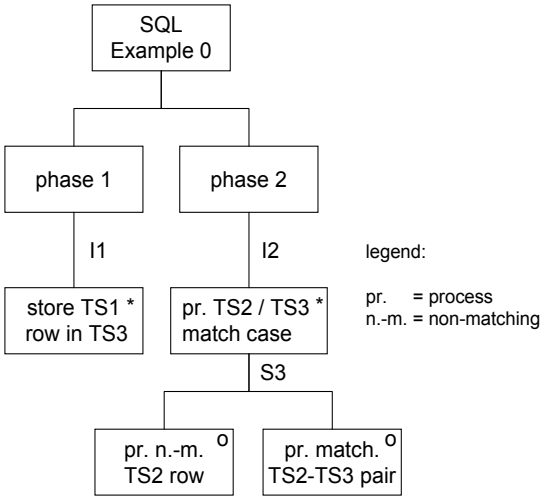


Abbildung 3.8: *Programmstruktur von „SQL Example 0" ohne Operationen*

Die Kontrollen lauten:

I1	TS1-Ende
I2	TS2-Ende
S3	Key der TS2-Zeile ist nicht in TS3 vorhanden

3.8.4 Problemlösung wegen der Kontrolle S3

Die nächste Schwierigkeit stellt sich ein, wenn es um die Auswahl und Zuordnung der Operationen geht. Wie soll das Programm feststellen, welcher S3-Fall vorliegt? Zunächst ist vorstellbar, dass für jede TS2-Zeile ein Leseversuch in TS3 stattfindet. Anhand des Ergebnisses kann S3 zweifelsfrei entschieden werden, aber das wäre eine Verschwendung von Ressourcen: Bei jedem INSERT oder UPDATE muss das RDBMS sowieso prüfen, ob die betreffende Zeile (bereits) existiert. Dafür muss es lesend auf die Tabelle TS3 zugreifen. Aus TS3 würde also doppelt gelesen, und das ist zugriffstechnisch teuer: Immerhin erfordert ja jeder Direktzugriff auf eine Tabelle mit Primärindex (oder beim Db2: Cluster Index), zunächst über den Indexbaum den Speicherort der Zeile zu lokalisieren und dann dort nachzusehen, ob es die Zeile gibt.

Verzichtet man aber auf den Leseversuch, kann man eigentlich nur noch raten, welcher Fall vorliegt. Genau das sollte das Programm nun tun, wobei das „Raten" darin besteht, einen INSERT- oder UPDATE-Versuch zu machen. Gelingt er, dann traf der angenommene Fall zu, und die Verarbeitung kann zur nächsten TS2-Zeile übergehen. Misslingt er, dann war die Annahme falsch und es muss die gegenteilige Zugriffsoperation stattfinden.

Dabei ist jedoch unschön, dass eine S-Kontrolle kein „Raten" vorsieht. Bislang war es immer so, dass eine Kontrolle dann entschieden werden konnte und musste, wenn sie „dran" war. Das ist beim „Raten" nicht möglich. Dieses Problem ist Michael A. Jackson natürlich auch aufgefallen und er hat dafür eine Lösung gefunden: das Backtracking. Dieses besteht darin, anstatt der Kontrolle eine plausible Annahme zu treffen und mit der Verarbeitung unter dieser Annahme fortzufahren, bis sie entweder bestätigt oder widerlegt wird. Falls sie widerlegt wird, muss ein geordnetes Verfahren den Übergang zur anderen Fallverarbeitung sicherstellen, wobei a) irreversible Operationen vermieden werden müssen, bis die Entscheidung für oder gegen die Annahme

sicher getroffen werden kann, und b) eventuell brauchbare Seiteneffekte des auf der Annahme basierenden Versuchs für die Alternativverarbeitung genutzt werden können.

Im vorliegenden Fall besteht eine plausible Annahme darin, dass TS2-Tupel den TS3-Tupel-Set im allgemeinen aktualisieren und seltener neue TS3-Einträge auslösen. Folglich sollte zuerst ein UPDATE-Versuch gemacht werden. Ausserdem führt der INSERT einer Zeile mit vorhandenem und eindeutigem Schlüssel zu einer Exception. Existiert die TS3-Zeile beim UPDATE-Versuch wider Erwarten nicht, dann folgt als programmierte Fehlerreaktion ein INSERT. In beiden Fällen folgt das Nachlesen. Das lässt sich als Flussdiagramm wie folgt darstellen:

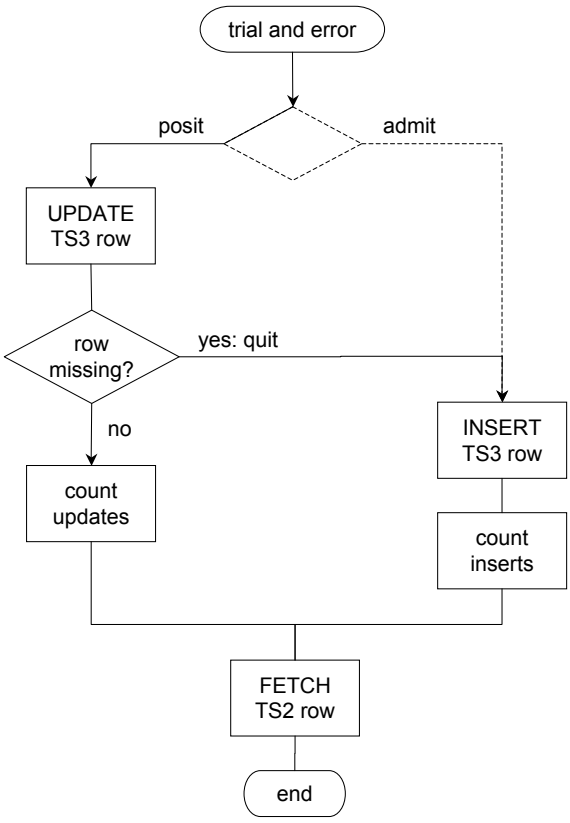


Abbildung 3.9: *Modellierung des „Ratens" als Backtracking-Lösung*

Hier tauchen neue Bezeichnungen und grafische Elemente auf:

- Das Schlüsselwort „posit" signalisiert, dass die Verarbeitung mit einer Annahme startet, hier also mit der begründeten Vermutung, dass eine TS2-Zeile im allgemeinen einen UPDATE in TS3 bewirken soll.
- „admit" bedeutet, zuzugeben, dass die Annahme im konkreten Fall falsch war.
- „quit" heißt, dass der gewählte Verarbeitungsstrang verlassen wird und ein „Sprung" zum alternativen Strang stattfindet.
- Die Raute unter „trial and error" bleibt leer. Ihre Kanten sind bis auf die linke obere Seite strichliert dargestellt, um zu visualisieren, dass es an dieser Stelle keine Kontrolle gibt.
- Dementsprechend ist die strichlierte Linie unter „admit" bis zum „Sprungpfeil" virtuell. Es gibt keinen solchen Kontrollfluss.

Lässt man die strichlierten Elemente und die neuen Schlüsselworte weg, dann sieht das Ganze nahezu wie eine einfache Fehlerreaktion nach fehlgeschlagenem UPDATE aus, die sich als einzelne Operation formulieren lässt. So wird es bestimmt auch immer wieder implementiert. Was soll daran falsch sein?

- Auf die Programmstruktur übertragen bedeutet ein solches Vorgehen, dass unter „pr(ocess) n(on)-m(atching) TS2 row" keine Operation angeordnet werden kann.
- Da S3 durch die Annahme ersetzt wird, die aktuelle TS2-Zeile bewirke einen UPDATE in TS3, ist die resultierende Programmstruktur nicht mehr auf die ihr zugrunde liegenden Datenstrukturen rückführbar, selbst wenn man die Strukturblöcke unter S3 vertauscht.
- Diese Art der Programmierung versteckt die Reaktion auf einen fachlich relevanten Fall in einem Stück Code zur Fehlerbehandlung, verschlechtert also die Transparenz.

Andererseits muss zugegeben werden, dass es übertrieben wäre, für eine solche Petitesse das große Rad des Backtracking zu drehen. Hier liegen doch auf beiden Seiten der Alternative S3 eigentlich nur einfach strukturierte, zusammengesetzte Operationen vor. Diese haben, wie im Flußdiagramm gezeigt, genau einen Eingang und einen Ausgang – sofern eine Alternative für „quit" gefunden werden kann. Das Backtracking ist aber für komplexe Fälle gedacht, bei denen die Ausstiegsentscheidung irgendwo „tief unten" im Strukturbaum getroffen werden muss und es dann unvermeidbar ist, den dieser Stelle folgenden Code zu übergehen. Das ist hier aber nicht der Fall.

Hier hilft uns die gute alte, vielfach verfluchte und verteufelte Flag-Programmierung. Sie ist wahrhaft ein Fluch, wenn mit Flags Fernsteuerung quer durch das Programm hindurch betrieben wird. Lokal angewendet und klar kontrolliert kann sie dagegen sehr hilfreich sein. Als Flussdiagramm dargestellt, ergibt sich mit einem einzigen Flag die folgende Lösung:

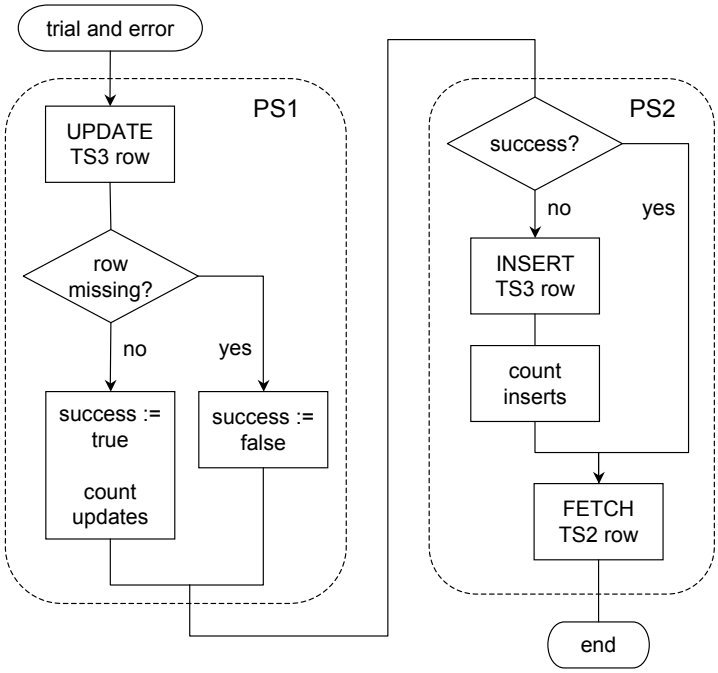


Abbildung 3.10: *Umwandlung in eine Procedure Statement-Sequenz*

Aus S3 und den „darunter" angeordneten Strukturblöcken ist auf diese Weise eine Sequenz aus PS1 und PS2 geworden. Das vereinfacht die Programmstruktur auf Kosten der Operationsliste.

3.8.5 Lösungsalternativen

Wozu diese Mühen? Gibt es nicht in Db2 das MERGE Statement und in MySQL das INSERT ... ON DUPLICATE KEY UPDATE Statement? MERGE mischt Zeilenmengen und erledigt somit die ganze Arbeit der zweiten Phase auf einmal, während die MySQL INSERT-Variante sich auf einzelne Zeilen richtet. Immerhin würde auch das den Aufwand vermindern, der mit der Ableitung der vorhergehenden Abbildung verbunden ist. Warum sich das Leben als Programmier(in) schwerer als nötig machen?

- Erstens geht es darum, für solche Fälle gangbare Lösungen aufzuzeigen, die sich auf *alle* RDBMS anwenden lassen, egal ob sie spezielle Abkürzungswege wie MySQL oder Db2 anbieten, oder das nicht tun.
- Zweitens gibt man mit mächtigen Statements wie MERGE die Kontrolle über die Ausführung und damit auch über die Performance komplett ab. Bei großen Zeilenmengen können ganz erhebliche Aufwendungen seitens des Db2 anfallen, die einen Speicherengpass und einen nachfolgenden Roll Back auslösen können, womit alles für die Katz wäre. Daher eignen sich solche Statements eher für Endbenutzer als für die professionelle Datenbankprogrammierung.

Wenn wir schon dabei sind, über Alternativen zur vorgeschlagenen Programmstruktur nachzudenken, dann sollte auch die erste Phase noch einmal betrachtet werden. Um der Inhalt einer Tabelle in eine andere Tabelle zu kopieren, gibt es mehrere Wege außer dem, dies Zeile für Zeile zu tun:

- Wohl die schnellste Kopiermethode außer hardware-unterstützter Replikation besteht darin, die Daten der Quelltabelle in eine sequentielle Datei zu entladen, diese gegebenenfalls zu sortieren, und sie dann in die Zieltabelle zu laden. Ist die Zieltabelle partitioniert, also in separate Speicherbereiche aufgeteilt, können sogar mehrere Partitionen parallel geladen werden. Daran ist allerdings sehr unschön, dass die Zeilendaten das Datenbanksystem verlassen, extern möglicherweise umgestellt oder angepasst und dann neu in das RDBMS eingeschleust werden. Währenddessen sind sie nicht durch die Mechanismen des RDBMS vor Datenverlust oder anderen potentiellen Gefahren geschützt.
- Sicher *und* schnell läuft das Kopieren ab, wenn Multi-Row Statements zum Lesen und Schreiben dienen. Solche Statements lesen und schreiben zahlreiche Zeilen „auf einmal", also beispielsweise 100 Zeilen, die bei Db2 in „host variable arrays" gehalten werden. Der Vorteil solcher Zugriffe besteht darin, den Kommunikationsaufwand zwischen dem Anwendungsprogramm und dem RDBMS zu minimieren. Allerdings bieten nur wenige RDBMSe diese Form an.
- Der dritte Weg besteht darin, dem RDBMS das Kopieren zu überlassen, indem eine spezielle Form der INSERT-Anweisung gewählt wird, der die Quelltabelle als Input dient. Diese Anweisung lautet in MySQL – und ähnlich auch in anderen RDBMS – im vorliegenden Fall wie folgt:

```
INSERT INTO TS3_0_Table (Tuple_Key, Tuple_Value)
SELECT Tuple_Key, Tuple_Value
  FROM TS1_0_View;
```

Danach ist die Tabelle TS3 mit den bekannten Werten aus TS1 gefüllt. Auch hier ist zu monieren, dass große Tabellen nicht auf diese Weise kopiert werden sollten, weil dies das jeweilige RDBMS überfordern könnte. Außerdem hat das Programm keinerlei Kontrolle über die Kopierfolge, was sich negativ auf den Aufbau der Zieltabelle nach dem Kopieren auswirken kann. Da MySQL keine Multi-Row Statements wie Db2 kennt und ein

Entwurfsziel der JSP/MSP-Modellierungen in Neutralität = Portabilität besteht, kommt keine der oben vorgestellten Alternativen in Frage.

Um die per INSERT mit Subselect angelegten TS3-Zeilen wieder loszuwerden, die ja erst durch das Programm SQL_Example0 geschrieben werden sollen, genügt eigentlich ein DELETE Statement ohne WHERE-Klausel:

```
DELETE FROM TS3_0_Table;
```

Daraufhin beschwert sich MySQL allerdings, im „safe update mode" seien Tabellenänderungen ohne Filterung durch WHERE nicht zulässig. Das ist ein Schutzmechanismus, der das versehentliche Ändern oder Löschen aller Zeilen einer Tabelle verhindern soll. Im aktuellen Fall stört er allerdings und muss temporär ausgeschaltet werden:

```
SET SQL_SAFE_UPDATES = 0;
DELETE FROM TS3_0_Table;
SET SQL_SAFE_UPDATES = 1;
```

Der Schutzmodus kann auch dauerhaft per Preference Panel ausgeschaltet werden, aber das ist nicht empfehlenswert. Nun sind alle Fragen geklärt, die bisher die Fertigstellung des Programmentwurfs verhinderten. Der nächste Schritt besteht darin, die Operationen aufzulisten, die Kontrollen fertigzustellen und die Programmstruktur damit zu bestücken.

3.8.6 Operationen, Kontrollen und Procedure Statements

Die Operationen lauten:

Op.	Bedeutung	Kommentar
1	TS1 Cursor öffnen	SELECT ohne Filterung oder Sortierung
2	TS2 Cursor öffnen	dito
3	ts1_rows_read := 0	TS1-Lese-Zähler initialisieren
4	ts2_rows_read := 0	TS2-Lese-Zähler initialisieren
5	FETCH TS1_row FROM TS1 Cursor	
6	ts1_rows_read := ts1_rows_read + 1	TS1-Lese-Zähler fortschalten
7	FETCH TS2_row FROM TS2 Cursor	
8	ts2_rows_read := ts2_rows_read + 1	TS1-Lese-Zähler fortschalten
9	TS3_row.Tuple_Key := TS1_row.Tuple_Key	
10	TS3_row.Tuple_Value := TS1_row.Tuple_Value	
11	INSERT TS3 row (Tuple_Key, Tuple_Value) FROM TS3_row	
12	ts3_inserts_from_ts1 := 0	INSERT-Zähler für Kopieren initialisieren
13	ts3_inserts_from_ts1 := ts3_inserts_from_ts1 +1	… und fortschalten
14	ts3_inserts_from_ts2 := 0	INSERT-Zähler für Aktualisieren initialisieren
15	ts3_inserts_from_ts2 := ts3_inserts_from_ts2 +1	… und fortschalten
16	ts3_updates_from_TS2 := 0	UPDATE-Zähler initialisieren

Op.	Bedeutung	Kommentar
17	ts3_updates_from_TS2 := ts3_updates_from_TS2 +1	… und fortschalten
18	UPDATE TS3 row SET TS3_row.Tuple_Value := TS3_row.Tuple_Value + TS2_row.Tuple_Value WHERE TS3_row.Tuple_Key = TS2_row.Tuple_Key	
19	success := `true`	Zeile wurde erfolgreich geändert
20	success := `false`	Zeile existiert nicht
21	TS3_row.Tuple_Key := TS2_row.Tuple_Key	
22	TS3_row.Tuple_Value := TS2_row.Tuple_Value	
23	TS1 Cursor schließen	
24	TS2 Cursor schließen	
25	„SQL Example 0“-Statistik ausgeben	
26	Stopp	

Darin werden die folgenden Strukturen mit identischem Aufbau verwendet:

```
TSx_row struct(                    // mit x = 1 | 2 | 3
    Tuple_Key      char(1);        // Key-Länge ist meistens größer
    Tuple_Value    num(s4)         // s = signed
);
```

In dieser Struktur werden die Attribute in Groß-Kleinschreibung benannt, weil es sich dabei um Spaltennamen in relationalen Tabellen handelt.

Die Kontrollen sind:

I1	TS1-Ende	
I2	TS2-Ende	
S4	TS3 row missing	(nach UPDATE-Versuch)
S5	success = `false`	

Die Procedure Statements lauten folgendermaßen:

```
seq PS1;
    8,18;
    if S4 then
        20
    else
        19,17
    fi
qes;
```

```
seq PS2;
    if S5 then
        21,22,11,15
    fi;
    7
qes;
```

3.8.7 Vorläufige Programmstruktur mit Operationen

Aus der „Abbildung 3.8: Programmstruktur von „SQL Example 0“ ohne Operationen“ auf Seite 36 und den Definitionen im Kapitel „3.8.6 Operationen, Kontrollen und Procedure Statements“ auf Seite 40 ff folgt die nachfolgend gezeigte – noch vorläufige – Programmstruktur von „SQL Example 0“.

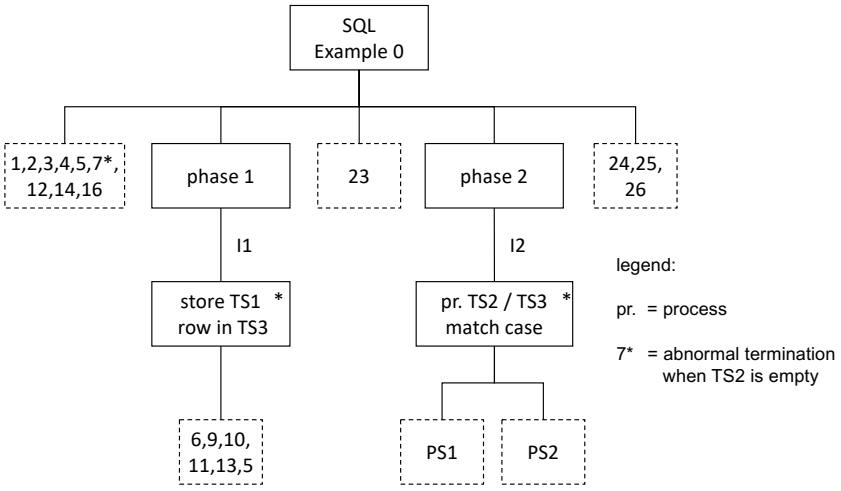


Abbildung 3.11: *Vorläufige Programmstruktur von „SQL Example 0“ mit Operationen*

Anmerkungen

- Die Tabellen selbst werden weder geöffnet noch geschlossen. Es werden lediglich zwei Lese-Cursors für TS1 und TS2 geöffnet, wobei dem RDBMS jeweils eine Query mitgegeben wird, die es zur Bestimmung der Ergebnismenge verwendet. Die Cursors werden geschlossen, sobald das Programm sie nicht mehr benötigt, also der erste Cursor nach der Phase 1 und der zweite Cursor nach der Phase 2.
- Diese Queries sehen lediglich das Lesen ohne Sortierung und ohne Filterung vor. Folglich werden die Zeilen in der physischen Folge angeliefert, in der das RDBMS sie antrifft.
- Das Öffnen des zweiten Cursors zu Beginn ist nicht zwingend erforderlich, aber das Programm kann dann frühzeitig feststellen, ob TS2 leer ist. In diesem Fall wäre der Lauf sinnlos und würde abgebrochen.
- An die Stelle von S3 und „pr(ocess) n(on)-m(atching) TS2 row“, sowie „pr(ocess) m(atching) TS2-TS3 pair“ ist die Sequenz aus PS1 und PS2 getreten. Das ist das erste Mal, dass die aus den Datenstrukturen abgeleitete Programmstruktur mit Operationen wegen einer Backtracking-Situation umgewandelt wird, wobei die Abstammung aus der Programmstruktur ohne Operationen weiterhin nachvollziehbar bleibt.

3.8.8 Implementierungsvorbereitungen

Im Ernst? Solch ein triviales Programm schreibt sich doch quasi von selbst. Eine einzige Klasse für die `main()` reicht doch völlig aus, um den Code darin als Monolithen unterzubringen. Wozu also mehr Aufwand treiben?

- Das so entstehende Programm wäre nur für MySQL brauchbar, weil sich die fachliche Logik und die Implementierungsdetails in einem Monolithen nicht voneinander trennen lassen. Die Portierung auf ein anderes RDBMS mit anderen Konventionen – wie beispielsweise Db2 – würde es nötig machen, das gesamte Programm umzuschreiben, obwohl sich an der fachlichen Logik nichts ändert.
- Folglich ist es nötig, die MySQL-spezifischen Implementierungsdetails von der fachlichen Logik zu trennen. Das kann nur durch Aufteilung auf mehrere Instanzen erreicht werden. Analog zu den File Services bietet es sich an, SQL Services für MySQL als eigenes Package zu entwickeln, das in SQL_Example0 importiert wird. Dieses Vorgehen ermöglicht die Portierung auf ein anderes RDBMS ohne Änderung des fachlichen Codes, indem das an MySQL orientierte SQL Service Package an dessen Konventionen angepasst oder völlig neu geschrieben wird. Die Namen der Zugriffsmethoden sollten auch daher neutral gehalten sein.
- Da hier erstmalig ein RDBMS-basiertes Programm entwickelt wird, sind noch einige Details zu klären, die sich auch auf die Operationsliste auswirken können. Darum geht es im folgenden.

3.8.9 Verbindungsaufbau zum RDBMS

Db2-Programmierer(innen) im Großrechnerumfeld müssen sich nicht damit beschäftigen, wie die Verbindung zwischen ihrem Programm und dem Db2 aufgebaut wird. Das übernimmt das Trägersystem des Programms, also beispielsweise TSO, IMS TM® oder CICS®. Es erledigt auch die Authentifizierungsoperationen gegenüber dem Db2. RDBM-Systeme wie MySQL verlangen dagegen von jedem Anwendungsprogramm, sich selbst zu authentifizieren. Das sieht beispielsweise wie folgt aus, wobei aufgrund höherer Flexibilität die Entscheidung für die Nutzung von `MysqlDataSource` anstelle des `DriverManager` fiel:

```
MysqlDataSource ds = new MysqlDataSource();
ds.setURL("jdbc:mysql://localhost:3306/tuple_db");
ds.setUser("msp_admin");
ds.setPassword("<password>");

// connect to the MySQL server
try (Connection con = ds.getConnection();
// more try code t.b.d.
{
// access code t.b.d.
}
catch (SQLException ex)
{
   System.out.println("++++ SQL exception" );
   System.out.println("SQLSTATE = " + ex.getSQLState());
   Logger lgr = Logger.getLogger(TestFramework.class.getName());
   lgr.log(Level.SEVERE, ex.getMessage(), ex);
}
```

Dieser Code stammt aus dem Umfeld des internationalen Schulsportwettbewerbs auf MySQL-Basis, der im Band 5-C2 beschrieben wird. Dort heisst die Database `issc_db` und ihr Administrator ist der `issc_admin`.

Gegen dieses Beispiel kann zu Recht eingewendet werden, dass es nicht klug ist, Login-Merkmale in einem Programm explizit offenzulegen. Eine Alternative besteht darin, hierfür Property-Daten zu verwenden, die aus einer Datei bereitgestellt werden. Das macht die Sache aber nicht prinzipiell besser. In Umgebungen mit stark schwankender Last kommt hinzu, dass ein Programm seine Datenbankverbindungen nicht – mit Hilfe des `DriverManager` – selbst aufbauen sollte, sondern sich auf das Connection Pooling durch eine `DataSource` stützen sollte. Diese ermöglicht auch die jeweils erforderliche Authentifizierung mittels JINDI (Java™ Naming and Directory Interface). Da Connection- und Berechtigungsfragen hier jedoch nicht im Zentrum stehen, dürfen diese Themen ausgeklammert bleiben.

Für die Programmierung der SQL Services ist jedenfalls festzuhalten, dass die Zugriffsoperationen an das Handle `con` anknüpfen und dieses folglich im Kernbereich der Services gehalten werden muss. Um die Services auch für Spezialfälle nutzen zu können, ist es jedoch vorstellbar, dieses Handle exportierbar zu machen.

3.8.10 Übergabe der DML an das RDBMS

Wer bisher nur mit statischer SQL unter Db2 und COBOL II gearbeitet hat, wird sich erst einmal – zumindest hinsichtlich der prinzipiellen Vorgehensweise – mit dynamischer SQL befassen müssen, um den Umstieg auf RDBM-Systeme wie MySQL gedanklich vorzubereiten. Diese gehen nämlich nicht von vorkompilierter SQL aus, sondern interpretieren die ihnen übergebene SQL jeweils neu. Ein Beispiel sieht wie folgt aus:

```
String query         = "SELECT Language_Code, Country_Code, Region_ID, " +
                       "Region_Name FROM Region_Language_Table";
String  languageCode = null;
String  countryCode  = null;
char read_stat       = 'D';

// establish the connection (see previous code snippet)

// "more try code t.b.d." is replaced by:
PreparedStatement pst = con.prepareStatement(query,
                        ResultSet.TYPE_SCROLL_SENSITIVE,
                        ResultSet.CONCUR_UPDATABLE);
ResultSet         rs  = pst.executeQuery();
```

Die Optionen der Methode `prepareStatement()` werden an dieser Stelle nicht erklärt. Wichtig ist hier allein, dass auf der Basis der neu eröffneten Connection der DML-String `query` per `prepareStatement()` an MySQL übergeben und dort sowohl syntaktisch geprüft als auch für die Ausführung vorbereitet, also kompiliert wird. Die Methode `executeQuery()` stellt die angeforderten Daten für den Lesezugriff bereit.

3.8.11 Lesezugriffe per Cursor

Ein Cursor ist ein virtueller Zeiger, der nach dem Bereitstellen einer Ergebnismenge zunächst vor deren erste Zeile weist. Die Methode `next()` bewegt den Cursor, indem sie die Zeilen aus der Menge, die durch das Handle `rs` repräsentiert wird, beispielsweise wie folgt abruft:

```
// "access code t.b.d." is replaced by:
// read 1st line of result set
// Result set empty?
if (rs.next() == false)
{
   System.out.println("++++ Result set empty" );
   return;
}
[…] (elided)
// scan region_language_table
while (read_stat == 'D')
{
   // check region_language row
   languageCode = rs.getString("Language_Code");
   countryCode  = rs.getString("Country_Code");
   […]
   // read next line of result set
   if (rs.next() == false)
      { read_stat = 'E'; }
}
```

Die obere Abfrage

```
if (rs.next() == false)
```

beschafft die erste Zeile und prüft, ob diese existiert. Beim Ergebnis `false` ist die Menge leer. Das Ergebnis `true` bedeutet, dass der Zugriff erfolgreich war. Der Cursor steht dann auf der ersten Zeile, deren Inhalte ausgelesen und gegebenenfalls modifiziert werden können. Ausserdem kann diejenige Zeile, auf welche der Cursor zeigt, gelöscht werden, ohne dafür ein DML-Statement aufbauen und kompilieren zu müssen.

Die Zeilendaten werden im Beispiel-Code mit `getString()` beschafft, wobei der jeweilige Spaltenname als Parameter dient. Alternativ kann die Spaltennummer vorgegeben werden, was aber als höchst fehleranfällig erscheint. Für jeden Datentyp, der in einer MySQL-Tabelle auftreten kann, ist eine eigene Getter-Methode – und parallel dazu eine Setter-Methode – vorgesehen. Übrigens ruft die oben gezeigte Query vier Spalten ab, wovon nur zwei Werte explizit in programminterne `String`-Variable übernommen werden, weil der mit […] ausgeklammerte Code damit arbeitet. Die anderen beiden Strings werden direkt per `getString()` an eine Ausgabemethode übergeben.

Somit zeichnet sich für die cursor-basierten Zugriffe eine Hierarchie mit vier Ebenen ab:

- Die oberste Ebene ist die der Connection. Ohne erfolgreichen Verbindungsaufbau ist kein Zugriff möglich.
- Auf der zweiten Ebene wird die DML übergeben und kompiliert = interpretiert, sowie gegebenenfalls der Result Set bereitgestellt.

- Auf der dritten Ebene erfolgt der sequentielle Zeilenabruf per Cursor-Zugriff.
- Die vierte Ebene handhabt die Daten in den Spalten der jeweils aktuellen Zeile, also derjenigen, auf welcher der Cursor positioniert ist.

Diese Struktur wird sich auch im Aufbau des diesbezüglichen Codes der SQL Services widerspiegeln. Die zweite bis vierte Ebene dient im allgemeinen Anwendungsfall dem Zugriff auf mehrere Tabellen. Sie muss daher für die Existenz parallel aufgesetzter Queries vorbereitet werden. Durch geeignete Kapselung der Daten und Methoden, sowie durch möglichst RDBMS-neutrale Benennungen kann ein hohes Maß an Unabhängigkeit von der jeweiligen „Physik" erreicht werden.

3.8.12 UPDATEs in TS3

Das Programm SQL_Example0 versucht in der Phase 2 bekanntlich für jede TS2-Zeile, damit zuerst einen TS3 UPDATE vorzunehmen, dessen Aufgabe es ist, den TS2-`Tuple_Value` zum TS3-`Tuple_Value` zu addieren. Dafür gibt es bei MySQL drei Möglichkeiten:

- Es kann ein UPDATE Statement aufbauen, das explizit den TS2-`Tuple_Value` und den TS3-`Tuple_Key` vorgibt. Dieses Statement muss anschliessend präpariert und mit `executeUpdate()` ausgeführt werden. Diese Methode liefert einen `int`-Wert zurück, der angibt, wieviele Zeilen geändert wurden.
- Es gibt einmalig ein UPDATE Statement mit Platzhaltern für `Tuple_Value` und `Tuple_Key` vor und lässt es mit `prepareStatement()` vorkompilieren. Platzhalter sind Fragezeichen anstelle konkreter Werte. Für jede TS2-Zeile ersetzt es die Platzhalter mit deren Spalteninhalten und lässt das Statement ausführen.
- Es wählt die zu ändernde Zeile per SELECT aus, der i.a. parametrisiert ist, eröffnet einen Cursor und führt damit die Änderung aus.

Es liegt auf der Hand, dass die dritte Variante zwar funktionieren würde, aber zugleich die aufwendigste und umständlichste wäre. Ausserdem war es doch ein Ziel der Phase 2, die zu ändernden TS3-Zeilen *nicht* einlesen zu müssen. Also scheidet diese Variante sofort aus.

Am Beispiel der TS2-Zeile mit dem `Tuple_Key 'C'` und dem `Tuple_Value 13` würde bei der ersten Variante das folgende Statement erzeugt, kompiliert und ausgeführt:

```
update TS3_0_Table set Tuple_Value = Tuple_Value + 13 where Tuple_Key = 'C'
```

Das wäre erheblich aufwendiger, als zunächst – gemäß der zweiten Variante – das folgende Statement vorkompilieren zu lassen

```
update TS3_0_Table set Tuple_Value = Tuple_Value + ? where Tuple_Key = ?
```

und anschließend für jeden UPDATE-Versuch die Platzhalter durch die jeweiligen TS2-Zeilenwerte zu ersetzen. Hier fehlt übrigens das abschließende Semikolon, weil es für die Programm-Interaktion mit MySQL nicht benötigt wird. Erst wenn mehrere Statements gemeinsam übergeben werden, ist das Semikolon als Trenner erforderlich. Für den `Tuple_Key 'C'` lauten die Zuweisungen wie folgt:

```
ps.setInt(1, 13);
ps.setString(2, "C");   // JDBC type CHAR = Java type String
```

Die Ordinalzahlen 1 und 2 geben die Position des jeweiligen Platzhalters an.

Dieses Vorgehen ist offensichtlich das schnellste und effizienteste der drei Varianten. Die Operation 18 geht allerdings von der ersten – oben erläuterten – UPDATE-Variante aus. Das ist noch zu ändern. Dagegen muss die programminterne Zeilenzählung nicht zwingend geändert werden, weil jeweils nur eine oder keine Zeile geändert wird – letzteres dann, wenn die zu ändernde TS3-Zeile nicht existiert.

Es kann aber zumindest nicht schaden, in den SQL Services die Anzahlen der jeweils betroffenen Zeilen sowohl aufzukumulieren als auch – für eine aussagekräftigere Abbruch-Information – den jeweils letzten rückgemeldeten Wert aufzubewahren. Ein Abgleich der Statistiken am Lauf-Ende zeigt, ob den SQL Services in dieser Hinsicht vertraut werden kann. Künftige SQL-Beispielprogramme können sich dann auf die in den Services geführten Zähler verlassen, anstatt die Zeilen selbst zählen zu müssen.

3.8.13 INSERTs in TS3

Analog zu UPDATE stehen Varianten mit und ohne Platzhalter für diese INSERTs zur Verfügung. MySQL bietet allerdings eine weitere Möglichkeit, die es ermöglicht, Einfügungen per Cursor vorzunehmen, was beispielsweise Db2 nicht „kennt". Dazu ist es erforderlich, zunächst einen Cursor zu öffnen. Dieser Cursor wird auf die „Staging Area" gestellt, die man sich als einen Montageplatz für Zeilen vorstellen kann. Dort wird die jeweilige Zeile mit ihren Inhalten oder auch mit NULL-Werten bestückt und schließlich explizit eingefügt. Es ist möglich, während der Verarbeitung einer Ergebnismenge zwischendrin eine oder mehrere Zeilen darin einzufügen und anschließend mit `moveToCurrentRow()` zu derjenigen Zeile zurückzukehren, auf welcher der Cursor vor dem Einfügen positioniert war.

Cursor-basierte INSERTs sind eine attraktive Option, weil sie ohne die positionalen Platzhalter auskommen, die bei umfangreichen Statements durchaus zu Fehlern Anlass geben können. Das ist im vorliegenden Fall jedoch nicht zu befürchten; zudem erscheint der Aufwand für eine extra dafür erforderliche Query hierbei als übertrieben. Wird eine Datenmenge aber sowieso per Cursor bearbeitet, dann kann das – anfänglich als ungewöhnlich erscheinende – Vorgehen durchaus attraktiv sein. An dieser Stelle darf offen bleiben, ob neu eingefügte Zeilen beim Durcharbeiten einer Ergebnismenge, die unmittelbar aus einer Basistabelle stammt, dem sie erzeugenden Programm später erneut „begegnen" können. Dieses potentielle Problem kann bezüglich TS3 nicht auftreten.

In der Phase 1 werden die TS1-Zeilen nach TS3 kopiert. Dazu gehört die Operation 11. Sie lautet „ausgeschrieben" am Beispiel der Zeile mit dem `Tuple_Key 'A'` wie folgt:

```
insert into TS3_0_Table (Tuple_Key, Tuple_Value) values ('A', -3)
```

Mit Platzhaltern sieht das folgendermaßen aus:

```
insert into TS3_0_Table (Tuple_Key, Tuple_Value) values (?, ?)
```

Die Zuweisungen zu den Platzhaltern lauten im vorliegenden Fall

```
ps.setString(1, "A");
ps.setInt(2, -3);
```

… also hinsichtlich der Positionsnummern genau umgekehrt. Auch das ist in der Operationsliste noch zu ergänzen. Die Operation 7 mit Platzhaltern funktioniert für beide INSERTs. Die mit den Setter-Methoden zu übertragenden Inhalte stammen dann aus TS2.

In den beiden vorangehenden Code-Beispielen wird übrigens immer derselbe Handle-Name `ps` verwendet, was unter Garantie zu Durcheinander führt. Die Regel lautet daher: Jedes durch `prepareStatement()` er-

zeugte Handle muss einen eigenen Namen – oder einen neuen Array-Eintrag im Fall der Verwaltung multipler Handles per Array – bekommen.

Eine weitere Schlussfolgerung für die SQL Services liegt nahe: Jedem an sie übergebenen Statement ist ein eigener Speicher- und Zählerbereich zuzuordnen. Das Programm muss zu Beginn dessen Dimension vorgeben, wofür eine weitere Operation erforderlich ist. Zudem wird eine zur bisherigen Klasse `StreamCode` analoge Enumerator-Klasse `SQL_ID` benötigt, deren Elemente die Indizes für die Speicher- und Zählerbereiche darstellen. Der höchste Index in dieser Klasse muss gleich der Dimensionsvorgabe minus 1 sein, was beim Anlegen eines Statements geprüft wird.

3.8.14 Steuernde Klassen

Da SQL_Example0 ein triviales Programm ist, wenn die Zugriffsaufgaben in die SQL Services ausgelagert werden, bietet sich eine simple Anordnung der steuernden Klassen an, die sich an den bereits bekannten Beispielen aus dem Band 1 orientiert. Für jede Phase sollte es eine statische Process Control-Klasse geben, die von I1 beziehungsweise von I2 aktiviert wird.

3.8.15 Lese-Klasse(n)

Die JSP-typische Verteilung der Lesezugriffe auf die Programmstruktur mit Vorlesen und Nachlesen, sowie die Schleifenkontrollen I1 und I2 legen es nahe, die schon von der Verarbeitung sequentieller Dateien bekannte Konstruktion zu übernehmen. Dabei ist allerdings noch zu entscheiden, ob eine gemeinsame Utility-Klasse für die Tabellen TS1 und TS2 ausreicht, oder ob dafür zwei Klassen erforderlich sind, die allerdings nahezu identische Aufgaben haben. Tendenziell liegt es nahe, dafür eine einzige Klasse zu verwenden, in der die Lese-Ergebnis-Speicher für die beiden Tabellen als Array-Elemente behandelt werden.

Da keine Schlüsselvergleiche stattfinden, ist auch keine Klasse für die – konkatenierten – Zeilenschlüssel erforderlich.

3.8.16 Fertigstellung der Programmstruktur mit Operationen

Die Aufgabenteilung zwischen „SQL Example 0“ und den SQL Services, sowie das Verwenden von Platzhaltern ermöglichen es, die Operationsliste relativ kurz zu halten. Die neuen und die geänderten (kursiv geschriebenen) Operationen lauten wie folgt:

Op.	Bedeutung	Kommentar
9	*Key-Platzhalter für TS3 INSERT := TS1_row.Tuple_Key*	
10	*Value-Platzhalter für TS3 INSERT := TS1_row.Tuple_Value*	
11	*INSERT INTO TS3 (Tuple_Key, Tuple_Value) values (?, ?)*	*mit ersetzten Platzhaltern*
18	*UPDATE TS3 SET Tuple_Value = Tuple_Value + ? WHERE Tuple_Key = ?*	*mit ersetzten Platzhaltern*
21	*Key-Platzhalter für TS3 INSERT := TS2_row.Tuple_Key*	
22	*Value-Platzhalter für TS3 INSERT := TS2_row.Tuple_Value*	

Op.	Bedeutung	Kommentar
27	Connection öffnen & SQL Array-Größe vorgeben	
28	Connection schließen	
29	TS1 SELECT kompilieren	
30	TS2 SELECT kompilieren	
31	TS3 INSERT kompilieren	
32	TS3 UPDATE kompilieren	
33	Key-Platzhalter für TS3 UPDATE := TS2_row.Tuple_Key	
34	Value-Platzhalter für TS3 UPDATE := TS2_row.Tuple_Value	

Die Kontrollen sind unverändert. Die Procedure Statements lauten jetzt folgendermaßen:

```
seq PS1;
    8,33,34,18;
    if S4 then
        20
    else
        19,17
    fi
qes;

seq PS2;                // unverändert
    if S5 then
        21,22,11,15
    fi;
    7
qes;
```

Die vervollständigte Programmstruktur mit Operationen folgt als „Abbildung 3.12: Finale Programmstruktur von „SQL Example 0“ mit Operationen“ auf Seite 50:

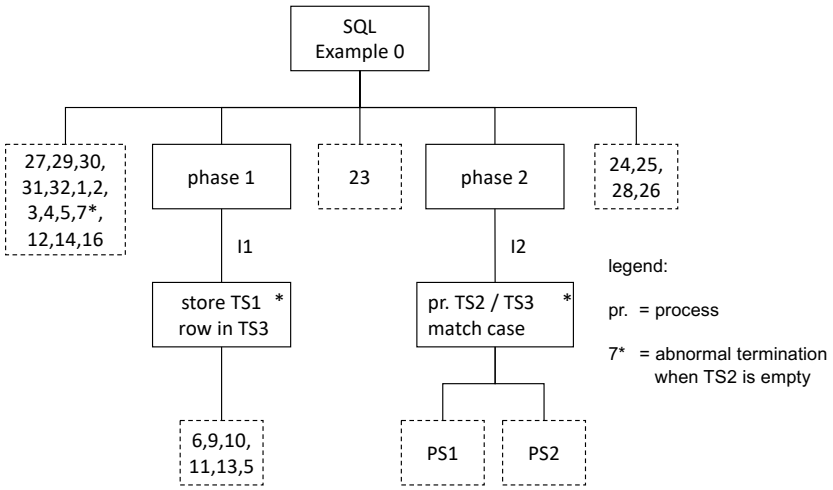


Abbildung 3.12: *Finale Programmstruktur von „SQL Example 0" mit Operationen*

3.8.17 Klassendiagramm von SQL_Example0

Damit sind die Implementierungsvorbereitungen weitgehend abgeschlossen. Die noch ausstehenden Detail-Entscheidungen über die Klassenstruktur erfordern keine methodischen Überlegungen, sondern können angesichts der Vorarbeiten im Band 1 während des Implementierens getroffen werden.

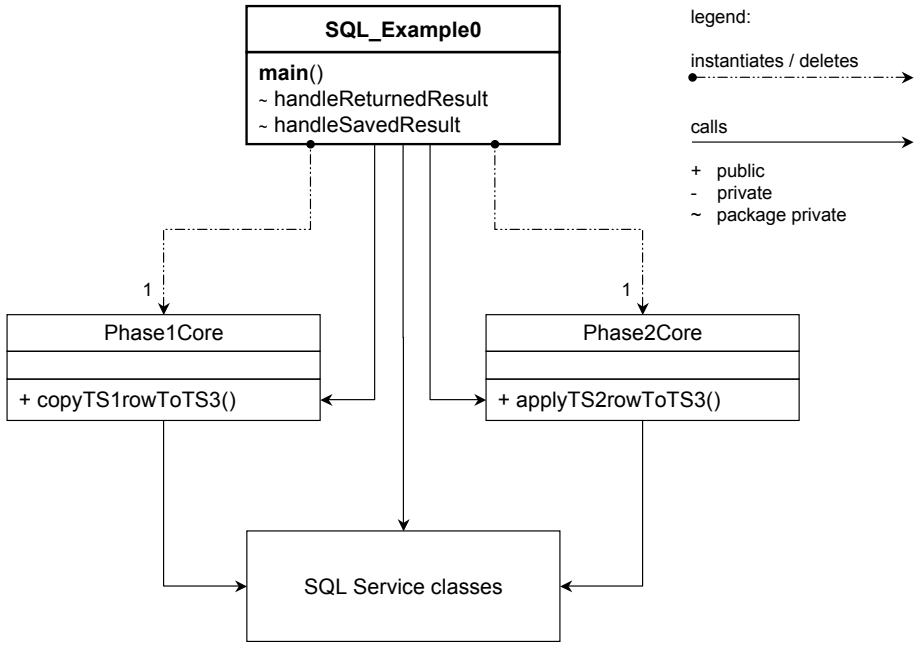


Abbildung 3.13: *Klassendiagramm von SQL_Example0*

3.9 Implementierung von SQL_Example0 in Java

Die Implementierung von SQL_Example0 auf der Basis der SQL Services und die zugehörigen Testberichte sind im Band-4-B1-Kapitel „SQL_Example0 in Java“ enthalten. Das neue Programm lief auf Anhieb korrekt und erzeugte das folgende Laufprotokoll:


```
*** SQL_Example0 ***
Trace settings
Display method names   : false
Display detail data    : false

SQL_Example0 Statistics
=== Input (internal variables) ===
TS1 rows read        = 8
TS2 rows read        = 7
=== Output ===
TS3 inserts from TS1 = 8
TS3 inserts from TS2 = 2
TS3 updates from TS2 = 5
=== SQL Service variables ===
TS1 rows read         = 8
TS2 rows read         = 7
TS1 SELECT exec calls = 1
TS2 SELECT exec calls = 1
TS3 INSERT exec calls = 10
TS3 UPDATE exec calls = 7
*** Successful completion ***
```

Die MySQL Shell zeigt die – korrekte – Ergebnismenge wie folgt an:

```
table TS3_0_Table;
+-----------+-------------+
| Tuple_Key | Tuple_Value |
+-----------+-------------+
| A         |          -3 |
| C         |          19 |
| F         |         -12 |
| H         |           9 |
| I         |          -2 |
| K         |          11 |
| N         |          25 |
| R         |          22 |
| T         |           7 |
| X         |           5 |
+-----------+-------------+
10 rows in set
```

Die Zeilen sind also auch hier nach dem `Tuple_Key` aufsteigend geordnet.

Die Anzahlen in der Statistik erscheinen auf den ersten Blick hinsichtlich der UPDATE-Werte widersprüchlich zu sein. Das Programm meldet fünf `TS3 updates from TS2`, während die SQL Services sieben `UPDATE exec calls` ausweisen. Das liegt daran, dass in zwei Fällen, nämlich bei den TS2-Schlüsseln `'H'` und `'I'`, keine korrespondierenden TS1-Zeilen vorhanden sind. Die versuchten UPDATEs gehen daher ins Leere und es müssen stattdessen INSERTs vorgenommen werden.

3.9.1 Sind wir nun fertig?

Wäre SQL_Example0 ein Programm, das eine sequentielle Ausgabedatei erzeugt, dann würde die Antwort „Ja!“ lauten. Dieses Programm könnte man immer wieder laufen lassen, weil es seine Ausgabedatei jedes Mal überschreiben würde. Das gilt allerdings nur unter macOS® und anderen Desktop-Betriebssystemen. Auf einem IBM-Großrechner käme es dagegen darauf an, ob das Überschreiben durch Vorgaben in der steuernden Job Control Language (JCL) erlaubt ist, oder ob ein solcher Versuch abgewiesen würde.

SQL_Example0 ist aber kein solches Programm, denn es operiert ausschließlich auf relationalen Tabellen. Wird es nach dem Erzeugen der TS3-Inhalte erneut gestartet, dann passiert folgendes:

```
*** SQL_Example0 ***
Trace settings
Display method names   : false
Display detail data    : false
++++ Executing a prepared statement failed for SQL ID >2< in executeUpdate()
java.sql.SQLIntegrityConstraintViolationException: Duplicate entry 'A' for key
'ts3_0_table.PRIMARY'
       at com.mysql.cj.jdbc.exceptions.SQLError.createSQLException(SQLError.
java:117)
       at com.mysql.cj.jdbc.exceptions.SQLError.createSQLException(SQLError.java:97)
       at com.mysql.cj.jdbc.exceptions.SQLExceptionsMapping.translateException(SQLEx
ceptionsMapping.java:122)
       at com.mysql.cj.jdbc.ClientPreparedStatement.executeInternal(ClientPreparedSt
atement.java:953)
       at com.mysql.cj.jdbc.ClientPreparedStatement.executeUpdateInternal(ClientPrep
aredStatement.java:1092)
       at com.mysql.cj.jdbc.ClientPreparedStatement.executeUpdateInternal(ClientPrep
aredStatement.java:1040)
       at com.mysql.cj.jdbc.ClientPreparedStatement.executeLargeUpdate(ClientPrepare
dStatement.java:1340)
       at com.mysql.cj.jdbc.ClientPreparedStatement.executeUpdate(ClientPreparedStat
ement.java:1025)
       at sql_service.SQL_Util.executeUpdate(SQL_Util.java:442)
       at sql_example0.Phase1Core.copyTS1rowToTS3(Phase1Core.java:52)
       at sql_example0.SQL_Example0.main(SQL_Example0.java:111)
SQL Array Entry 0:
   SQL Statement: SELECT Tuple_Key, Tuple_Value FROM TS1_0_View
   Prepared Statement handle exists
   Result Set handle exists
   exec calls : 1
```

```
   row count  : 1
   read status: D
SQL Array Entry 1:
   SQL Statement: SELECT Tuple_Key, Tuple_Value FROM TS2_0_View
   Prepared Statement handle exists
   Result Set handle exists
   exec calls : 1
   row count  : 1
   read status: D
SQL Array Entry 2:
   SQL Statement: INSERT into TS3_0_Table (Tuple_Key, Tuple_Value) values (?, ?)
   Prepared Statement handle exists
   exec calls : 0
   row count  : 0
   read status:
SQL Array Entry 3:
   SQL Statement: UPDATE TS3_0_Table set Tuple_Value = Tuple_Value + ? where Tuple_
Key = ?
   Prepared Statement handle exists
   exec calls : 0
   row count  : 0
   read status:
Fatal error -500: ++++ Executing a prepared statement failed for SQL ID >2< in
executeUpdate()
/Users/.../Library/Caches/NetBeans/16/executor-snippets/run.xml:111: The following
error occurred while executing this line:
/Users/.../Library/Caches/NetBeans/16/executor-snippets/run.xml:68: Java returned:
12
BUILD FAILED (total time: 1 second)
```

Anmerkung

Die Angabe `SQL_Util.java:442` beruht auf dem unfertigen Stand der SQL Services. Durch Hinzufügung der noch fehlenden Methoden ist bei späteren Wiederholungen desselben Laufs eine andere Zeilennummer anstelle von `442` zu erwarten.

Aus Gründen der Übersichtlichkeit wurden die Exception-Stack-Ausgaben nach der mit ++++ beginnenden Fehlermeldung umgeordnet und als kompletter Textblock dargestellt. Aufgrund von konkurrierenden Ausgabe-Tasks ist das leider nicht der Fall. Danach folgt die Auflistung der Inhalte des SQL Array.

Der erneute Start von SQL_Example0 scheiterte also bereits beim ersten Versuch, eine bereits existierende Zeile erneut einzufügen. Das erscheint anfangs kein Problem zu sein, denn SQL_Example0 darf natürlich nur einmal laufen, um seine Aufgabe zu erfüllen. Was aber geschieht, wenn der erste Lauf von SQL_Example0 aus internen oder externen Gründen abbricht? Dann ist die TS3-Tabelle zwangsläufig unvollständig. Solche Abbrüche wurden absichtlich mit Hilfe der Debug-Funktion von NetBeans verursacht, wofür der TS3-Inhalt vorher komplett gelöscht wurde. Der Inhalt der Zieltabelle nach einem Abbruch in der Phase 1 ist dann beispielsweise wie folgt:

```
table TS3_0_Table;
+-----------+-------------+
| Tuple_Key | Tuple_Value |
+-----------+-------------+
| A         |          -3 |
| C         |           6 |
| F         |           2 |
| K         |          11 |
| N         |          17 |
+-----------+-------------+
5 rows in set
```

Es sind also nur fünf der acht TS1-Zeilen nach TS3 übertragen worden. Ein Abbruch in der Phase 2 ergibt beispielsweise den folgenden Tabellenzustand:

```
table TS3_0_Table;
+-----------+-------------+
| Tuple_Key | Tuple_Value |
+-----------+-------------+
| A         |          -3 |
| C         |           6 |
| F         |           2 |
| K         |          11 |
| N         |          25 |
| R         |          22 |
| T         |           7 |
| X         |           5 |
+-----------+-------------+
8 rows in set
```

Hier fehlen die Zeilen mit den Schlüsseln `'H'` und `'I'`. Zudem fand ein Teil der UPDATEs nicht statt.

3.9.2 Optionen nach einem Abbruch von SQL_Example0

Die einfachste Option nach einem Abbruch von SQL_Example0 besteht darin, die unvollständige Zeilenmenge in TS3 zu löschen und das Programm erneut zu starten. Wenn das Programm wegen eines internen Fehlers abbrach, dann wird dieser allerdings erneut auftreten. Also muss das Programm in diesem Fall zuerst repariert werden. Falls kein weiterer Fehler vorliegt, wird es anschließend die Tabelle korrekt füllen.

Dieses zugleich triviale und radikale Vorgehen lässt sich allerdings nur für kleine Datenmengen rechtfertigen. Wenn ein Datenbankprogramm nach Stunden abbricht, dann ist es eher ungut, dessen Lauf nach einer eventuellen Reparatur zu wiederholen, also die mühsam erzeugte Datenmenge zuvor komplett zu löschen. Das wäre nur dann zu rechtfertigen, wenn es sich nach dem Abbruch herausstellen sollte, dass diese Datenmenge wegen irgendwelcher Fehler defekt ist, und die Defekte nicht auf sanftere Weise behoben werden können. Im vorliegenden Fall ist SQL_Example0 nicht defekt, sondern sein Lauf wurde durch externe Eingriffe abgebrochen. Kann das geheilt werden? Vorweg: Nach einem Abbruch ist unklar, in welcher Phase er eintrat, es sei denn, der Übergang von der Phase 1 zur Phase 2 würde protokolliert und das Protokoll wäre vollständig.

- Wenn das Programm in der Phase 1 abbricht, dann sind nicht alle Zeilen aus TS1 nach TS3 übertragen worden. Das kann nachgeholt werden, indem in der neu gestarteten Phase 1 bei jeder TS1-Zeile geprüft wird, ob bereits eine TS3-Zeile mit demselben Schlüssel existiert. Ist das nicht der Fall, wird der INSERT nachgeholt. Es ist also ein Singleton SELECT erforderlich, der aus TS1 mit dem `Tuple_Key` bestückt wird, um in TS3 das Vorhandensein der korrespondierenden Zeile zu verifizieren. Das ist zwar umständlich, aber problemlos machbar.
- Ein Abbruch in der Phase 2 ist weitaus schwieriger zu kontern. Dann haben ja bereits UPDATEs und INSERTs anhand von TS2-Daten stattgefunden, wobei den Zeilen in TS3 nicht von vornherein „anzusehen" ist, ob sie aus TS1 oder aus TS2 stammen, und wie oft sie gegebenenfalls anhand von TS2-Daten manipuliert wurden. Daher müssen die in der Phase 2 bereits vorgenommenen Änderungen an TS3 rückgängig gemacht werden.
  - Das bedeutet, dass beim Übertragen von TS1 nach TS3 zusätzlich geprüft werden muss, ob die in TS3 gefundenen Zeilen unverändert sind, also denselben `Tuple_Value` aufweisen. Wenn das nicht zutrifft, muss der `Tuple_Value` aus TS1 per UPDATE in die korrespondierende TS3-Zeile übernommen werden.
  - Ausserdem müssen alle Zeilen, die ausschließlich aus TS2 stammen, gelöscht werden. Hierfür ist festzustellen, ob zu einem `Tuple_Key` aus TS2 derselbe `Tuple_Key` in TS1 existiert. Ist das der Fall, dann wurde die TS3-Zeile bereits gegebenenfalls rückgesetzt. Andernfalls ist die betreffende TS3-Zeile zu löschen, falls sie existiert, also nicht schon zuvor – wegen des mehrfachen Auftretens desselben Schlüssels in TS2 – gelöscht wurde.

Dieses Prozedere darf mit Recht als gruselig bezeichnet werden. Der Aufwand für die Entwicklung der neuen Datenstrukturen und für die Ableitung der erweiterten Programmstruktur dürfte beträchtlich sein. Dies sei uns eine Warnung, welche Probleme durch scheinbar triviale Datenbankprogramme geschaffen werden können.

3.9.3 Geht es nicht eleganter?

Das Genre der Abenteuerfilme lebt auch davon, dass ein scheinbar unausweichliches Verhängnis durch pfiffige Ideen abgewendet werden kann, wobei die Spannung der Zuschauer(innen) um so höher steigt, je aussichtsloser die Lage wird. Solche Ideen wurzeln oft darin, dass es gelingt, sich von dem aktuellen Dilemma zu lösen und einen ungewöhnlichen Gedanken zuzulassen. Hier liegt eine Parallele zum Problemlösungsverhalten in der Informatik, das in scheinbar festgefahrenen Situationen unerwartete Auswege finden und deren Potential auszuloten vermag. Was sind denn die eigentlichen Ursachen des zuvor geschilderten Wiederanlaufkrampfs?

- Die an der Verarbeitung beteiligten Tabellen werden nicht in einer definierten Reihenfolge be- und verarbeitet, sondern in zufälliger Folge – auch wenn MySQL die TS1- und TS3-Zeilen am `Tuple_Key` ausrichtet, was aufgabenseitig nicht gefordert ist und auch nicht genutzt wird.
- Daher kann es ohne zusätzliche Maßnahmen keinen „Aufsetzpunkt" geben, an dem die Verarbeitung nach einem Abbruch fortfahren kann, ohne bereits durchlaufene Schritte unabsichtlich zu wiederholen. Ebensowenig kann Vollständigkeit sichergestellt werden, da die zur Verarbeitung angebotenen Zeilen bei jedem Lauf in anderer Reihenfolge als zuvor erscheinen können.
- Selbst wenn die bislang durchlaufenen Schritte in einer weiteren Tabelle protokolliert würden, um damit später den Wiederanlauf zu steuern, könnte dieses Protokoll unvollständig oder inkonsistent sein. Das liegt daran, dass MySQL gewöhnlich im Modus „Autocommit" arbeitet, also jegliche Änderung, die ein Programm vornimmt, sofort „festschreibt". Je nachdem, welches Ausfallszenario eintritt, ist es in diesem Modus möglich, dass eine Änderung von TS3 zwar vorgenommen, aber nicht mehr protokolliert wird, weil das Programm genau dazwischen abbricht. Umgekehrt könnte zwar zuerst das Protokoll geschrieben und

dann die Änderung ausgeführt werden, aber ein Abbruch zwischen den beiden Aktionen würde ebenfalls einen inkonsistenten Zustand hinterlassen.

- Folglich besteht eine mögliche Problemlösung darin, den Autocommit-Modus abzuschalten und die Verarbeitung in Schritte aufzulösen, die entweder ganz oder gar nicht stattfinden. Das ist machbar, weil MySQL alle Änderungen zurücknimmt, die innerhalb eines solchen Schritts stattgefunden haben, wenn diese Änderungen nicht definitiv bestätigt werden. Diese Bestätigung heisst Commit.
- Nach alledem dürfte allerdings Einigkeit darüber bestehen, dass SQL_Example0 ein ungünstiges Demonstrationsobjekt für solche Bemühungen ist. Es gibt bessere Lösungen, die in den folgenden Kapiteln ausgelotet werden. Im Kapitel „8 Synchrones Mischen II“ auf Seite 233 ff folgt schließlich ein universell verwendbares Schema, das sich auch auf die Aufgabenstellung von SQL_Example0 anwenden lässt.
- Um Sie, geschätzte Leserinnen und Leser, nicht allzusehr auf die Folter zu spannen, nähern sich bereits die nun folgenden Kapitel dem Datenbankprogramm-Restart-Thema auf jeweils eigene Weise, auch um die mögliche Lösungsvielfalt zu demonstrieren.

Endnoten

1 Anmerkung des Autors: Die Installation der MySQL Command Shell erfordert zunächst einen Download von einer dafür eingerichteten Oracle-Seite (mit Authentifizierung als Oracle-Benutzer) für das jeweilige Betriebssystem und die anschliessende Installation. Unter macOS befindet sich danach das geladene Programm im Ordner /usr/local, beispielsweise als /usr/local/mysql-8.0.31-macos12-arm64. Um es zu starten, müssen im macOS-Terminal die folgenden Kommandos eingegeben werden:

```
echo 'export PATH=/usr/local/mysql/bin:$PATH' >> ~/.bash_profile
. ~/.bash_profile
mysql –u root –p
```

Das erste Kommando fügt den Pfad zu MySQL dem Bash Shell-Profil hinzu. Das zweite zwingt die Shell, das Profil neu zu laden. Mit dem dritten Kommando wird die MySQL Command Shell gestartet. Sie verbindet sich automatisch mit dem Server. Anschliessend wird das Root User-Passwort angefordert. Anstatt `root` kann auch ein anderer Administrator angegeben werden. Sobald die Anmeldung beendet ist, meldet sich MySQL mit dem Prompt

```
mysql>
```

Um mit einer bestimmten Database zu arbeiten, gibt man daraufhin beispielsweise

```
use tuple_db;
```

ein. Das vereinfacht die Namenshandhabung. Der MySQL Command Shell können beliebige SQL-Anweisungen gegeben werden, die mit Semikolon abzuschließen sind. Für andere RDBMS finden sich analoge Inbetriebnahmeanweisungen im Internet.