Eidgenössische Technische Hochschule Zürich
Ecole polytechnique fédérale de Zurich
Politecnico federale di Zurigo
Swiss Federal Institute of Technology Zurich

Dr. G. Alonso *C. Hagen, B. Kemme*

2. SCHNELLÜBUNG zu AR I

3. 7. 1997

Dauer: 60 Minuten

TAKE IT EASY!!

Name, Vorname (Druckschrift) ....

ETH-Nummer / Abteilung ....

Name (Unterschrift) ....

Aufgabe	Max. Punktzahl	Erreichte Punktzahl
1	25	
2	25	
3	25	
4	25	
Summe:	100	
	Note:	

Aufgabe 1: Advanced Transaction Models

a) **Geschlossen Geschachtelte Transaktionen (Closed Nested Transactions)**
Betrachten sie die folgenden Abläufe von geschlossen geschachtelten Transaktionen, wobei T_a und T_b Subtransaktionen von T_1 sind und $w_{a1}(x)$ und c_{a1} die Operationen der Subtransaktion T_{a1} von T_a sind. Geben Sie jeweils an, ob ein solcher Ablauf möglich ist, und begründen Sie Ihre Antwort. Ohne Begründung werde keine Punkte vergeben. (10 Punkte)

Lösung:

1. *Der Ablauf ist nicht möglich, da T_a die Sperre für x nicht freigibt, bzw. an T_1 weitergibt, bis sie beendet ist. In dem Szenario ist dies aber erst nach dem Zugriff von T_b auf x der Fall. Widerspruch.*
2. *Der Ablauf ist möglich, da nach dem Commit von T_{a1} die Sperre für x an T_a zurückgegeben werden kann, und nun T_a diese Sperre an T_{a2} weitergeben kann.*
3. *Der Ablauf ist je nach Interpretation von geschlossen geschachtelten Transaktionen möglich oder nicht möglich:*
   - *Der Ablauf ist möglich. Wenn T_{11} als direkte Subtransaktion von T_1 committed, gibt es die Sperre auf y an T_1 zurück. Wenn nun T_{a2} die Sperre anfordert, kann sie von T_1 über T_a an T_{a2} weitergegeben werden.*
   - *Der Ablauf ist nicht möglich. Die Elterntransaktion kann nur beim Start einer Subtransaktion Sperren an die Subtransaktion weitergeben. Da T_a bereits begonnen hat, bevor T_{11} beendet wird, kann T_1 die Sperre auf y nicht nachreichen.*
4. *Der Ablauf ist möglich, da überhaupt keine konfligierenden Operationen auftreten.*

b) **Flexible Transactions**
Der Aufbau von Flexible Transactions ist in der Vorlesung besprochen worden. Sie bestehen aus Pivot-Subtransactionen, kompensierbaren Subtransaktionen und wiederholbaren (retriable) Subtransaktionen, die nach gewissen Regeln hintereinander und parallel angeordnet sein können. Hier noch einmal eine Zusammenfassung von Regeln, wie Flexible Transactions angeordnet sein müssen:

1. Bevor die erste Pivot-Subtransaktion ausgeführt wird, dürfen nur kompensierbare Subtransaktionen ausgeführt werden.
2. Von jeder Pivot-Subransaktion aus gibt einen Ausführungspfad, der nur wiederholbare (retriable) Subtransaktionen enthält.
3. Zwischen zwei Pivot-Subransaktionen dürfen nur kompensierbare Subtransaktionen ausgeführt werden.

Zeigen Sie formal, dass eine derartig gebildete flexible Transaktion atomar ist, d.h. sie kann immer auf den Anfang zurückgesetzt werden (Rollback to Begin of Transaction) oder weiter fortgesetzt werden, bis das Commit erreicht wird. (15 Punkte).

Lösung:

Die Beschreibung der Konstruktion von Flexible Transactions war weder in der Vorlesung noch in der Prüfungsaufgabe ganz vollständig. Es fehlt die Bedingung, dass es von jeder wiederholbaren Subtransaktion aus ein Pfad geben muss, der nur wiederholbare Subtransaktionen enthält. Der Teil des Beweises, der zeigt, dass mit dieser Erweiterung der Konstruktion nach Ausführung einer wiederholbaren Subtransaktion die Transaktion immer erfolgreich beendet werden kann, ist ähnlich zu dem ensprechenden Beweisstück für Pivot-Subtransaktionen. Als Musterlösung werden wir jedoch den Beweis beschreiben, der zur vollen Punktzahl führte.

Wir notieren eine möglicherweise leere Menge von kompensierbaren Subtransaktionen mit $\mathbf{C}$*, eine möglicherweise leere Menge von wiederholbaren Subtransaktionen mit* $\mathbf{R}$ *und eine Pivot-Subtransaktion mit* P*.*

Beweis über Induktion über die Anzahl der Pivot-Subtransaktionen:

- *Induktionsbeginn: Die Transaktion hat eine Pivotsubtransaktion. Die Transaktion hat die Form* $\mathbf{C} - P - \mathbf{R}$*.*
  - *Sollte eine der Subtransaktionen aus* $\mathbf{C}$ *abbrechen, können die bereits ausgeführten Subtransaktionen kompensiert werden und die gesamte Transaktion wird auf den Anfang zurückgesetzt.*
  - *Werden alle Subtransaktionen aus* $\mathbf{C}$ *erfolgreich ausgeführt, aber* P *bricht ab, so können alle Subtransaktionen aus* $\mathbf{C}$ *kompensiert werden und die gesamte Transaktion wird auf den Anfang zurückgesetzt.*
  - *Werden alle Subtransaktionen aus* $\mathbf{C}$ *und* P *erfolgreich ausgeführt, dann kann die gesamte Transaktion erfolgreich beendet werden, da* $\mathbf{R}$ *nur wiederholbaren Subtransaktionen hat, die auf alle Fälle erfolgreich beendet werden können.*
- *Induktionsannahme: Eine Transaktion mit* n *Pivot-Subtransaktionen ist atomar.*
- *Induktionsschluss: Wir generieren eine Transaktion mit* $n+1$ *Pivot-Subtransaktionen. Diese ensteht aus einer Transaktion mit* n *Pivot-Subtransaktionen, indem nach Regeln 2 und 3 eine neue Pivot-Subtransaktion angehängt wird:* $- P_n - \mathbf{C} - P_{n+1} - \mathbf{R}$*:*
  - *Sollte eine der Subtransaktionen aus* $\mathbf{C}$ *abbrechen, können die bereits ausgeführten Subtransaktionen aus* $\mathbf{C}$ *kompensiert werden und von* P_n *aus kann der existierende alternative Pfad mit nur wiederholbaren Subtransaktionen immer erfolgreich ausgeführt werden.*

  - *Werden alle Subtransaktionen aus* $\mathbf{C}$ *erfolgreich ausgeführt, aber* P_n *bricht ab, so können alle Subtransaktionen aus* $\mathbf{C}$ *kompensiert werden und von* P_n *aus kann der existierende alternative Pfad mit nur wiederholbaren Subtransaktionen erfolgreich beendet werden.*
  - *Werden alle Subtransaktionen aus* $\mathbf{C}$ *und* P *erfolgreich ausgeführt, dann kann der von* P_{n+1} *aus existierende Pfad mit nur wiederholbaren Subtransaktionen erfolgreich beendet werden.*

Aufgabe 2: Atomarität in verteilten Systemen

Entwerfen Sie ein Protokoll, um persistente Nachrichten zu implementieren. Dieses Protokoll garantiert, dass jede Nachricht, die von Prozess p zu Prozess q gesendet wird, irgendwann bei q ankommt. Es können sowohl Kommunikationsfehler zwischen p und q auftreten als auch Knoten ausfallen. Fehler werden aber nach endlich langer Zeit behoben. Das Protokoll soll garantieren, dass nach der Recovery eines Fehlers alle Nachrichten empfangen werden, die gesendet worden waren, wegen dem Fehler aber nicht empfangen wurden (Dabei wird angenommen das lange genug kein erneuter Fehler auftritt). Prüfen Sie die Korrektheit Ihres Protokolls für die verschiedenen Fehlerfälle. (25 Punkte)

Lösung:

Eine mögliche Lösung des Problems besteht in der Verwendung persistenter Warteschlangen (persistent queues) zur Zwischenspeicherung der Nachrichten. Diese lassen sich folgendermassen implementieren:

Jeder Knoten A *besitzt für jeden anderen Knoten* B *eine Warteschlange* q_B *und ein Flag* $down_B$*. Das Flag wird gesetzt, wenn der entsprechende Knoten nicht erreichbar ist. Die Warteschlange ist eine Datei, in der Nachrichten zwischengesprichert werden können. Das Senden einer Nachricht von Rechner* A *zu Rechner* B *kann in einem von zwei Modi stattfinden. Im* Normalmodus *(er wird gewählt, wenn* $down_B$ *auf* A *nicht gesetzt ist) wird die Nachricht direkt an* B *gesendet.* B *sendet eine Bestätigung zurück. Falls diese Bestätigung ausbleibt (Timeout), wird angenommen, dass* B *ausgefallen oder nicht erreichbar ist. In diesem Fall wird* $down_B$ *auf* true *gesetzt und die Nachricht wird im* Fehlermodus *noch einmal verschickt. Im* Fehlermodus*, der immer dann gewählt wird, wenn* $down_B$ *gesetzt ist, wird die Nachricht in die lokale Warteschlange* q_B *eingefügt.*

In der Recovery-Phase eines Knotens, der ausgefallen war, sendet dieser alive*-Nachrichten an alle anderen Knoten. Ein Knoten* A*, der eine solche Nachricht von* B *erhält, sendet alle Nachrichten in* q_B *an* B*. Dabei wird für jede Nachricht die folgende Reihenfolge eingehalten:*

1. *Lese erste Nachricht* m *aus der Warteschlange*
2. *Sende* m *an* B

3. *Warte auf Bestätigungsnachricht von B*

4. *Entferne m aus der Warteschlange*

B "merkt sich", welche Nachrichten schon angekommen sind. Dazu werden z.B. die Identifikatoren der erhaltenen Nachrichten persistent gespeichert. Empfängt er eine Nachricht zum zweiten Mal, so sendet er ein Acknowledge und vernichtet dann die Nachricht. Der Algorithmus realisiert ein (lineares) 2-Phase-Commit-Protokoll zwischen den beiden Knoten, da Nachrichten auf A erst gelöscht werden, wenn sie garantiert auf B angekommen sind. Nachdem alle Nachrichten gesendet wurden, sendet A eine recovered*-Nachricht an B und setzt* $down_B$ *auf* false. *Ein Knoten hat seine Recovery-Phase beendet, wenn er von allen Knoten* recovered*-Nachrichten erhalten hat.*

Korrektheit des Protokolls:

Fällt der Empfänger vor *dem Versand einer Nachricht im Normalmodus aus, so schaltet der Sender in den Fehlermodus und schreibt die Nachricht in die lokale Warteschlange. Die Nachricht wird dann gesendet, sobald der Empfänger wieder erreichbar ist. Fällt der Empfänger* während *des Versands einer Nachricht aus, so kann er kein Acknowledge mehr senden und der Sender geht wiederum in den Fehlermodus. Fällt der Empfänger während einer Recovery-Phase aus, so bleiben beim Sender alle Nachrichten erhalten, die noch nicht bestätigt wurden. In der nächsten Recovery-Phase werden diese dann zugestellt. Kommunikationsfehler werden genauso behandelt wie Knotenausfälle. Wir gehen davon aus, dass die Wiederherstellung einer gestörten Kommunikationsverbindung von den betroffenen Knoten erkannt werden kann, sodass sie die Recovery starten können. Senderausfälle werden toleriert, da sich der Empfänger merkt, welche Nachrichten er schon erhalten hat. Ein Sender wird also schlimmstenfalls eine Nachricht mehrmals senden, falls ein Acknowledgement nicht mehr angekommen ist. Dies wird aber beim Empfänger erkannt.*

Bemerkung: Diese Protokoll blockiert, falls während der Recovery von B ein anderer Knoten K nicht läuft.

Aufgabe 3: Benchmarks

a) Was messen Benchmarks, die für Datenbanksysteme konzipiert sind? (5 Punkte)

Lösung:

- *Mit einem Benchmark wird die Leistungsfähigkeit der Teilsysteme eines Datenbanksystems (Hardware wie Prozessoren, Speicher, Terminals und Software wie das Datenbankverwaltungssystem uun das Betriebssystem) und deren Zusammenspiel gemessen.*
- *Es wird der Durchsatz an Transaktionen pro Zeiteinheit gemessen, der vom Datenbanksystem erreicht werden kann, wenn Zeitbeschränkungen für die Antwortzeit gegeben werden.*
- *Es werden die Kosten pro erreichte Transaktion pro Zeiteinheit angegeben (d.h. die Kosten in Abhängigkeit vom Durchsatz).*
- *Es wird getestet, ob das System die ACID-Eigenschaften (Atomarität, Konsistenz, Isolation und Dauerhaftigkeit einhält.*

b) Beschreiben Sie die Unterschiede zwischen dem TPC-C und dem TPC-D Benchmark. Für welche Applikationen sind sie jeweils konzipiert und was sind die Leistungsmetriken? (5 Punkte)

Lösung:

- *TPC-C misst OLTP-Arbeitslasten (Online Transaction Processing), bei denen der Hauptteil der Transaktionen aus wenigen, häufig modifizierenden Operationen besteht. Viele BenutzerInnen greifen gleichzeitig und interaktiv auf die Datenbank zu und erwarten kurze Antwortzeiten. Es beschreibt also das "Tagesgeschäft" einer Datenbankanwendung. (Kauf/Verkauf von Aktien, Tagesgeschäft einer Bank, Reisebuchungen etc.) Die Leistungsmetriken sind der Durchsatz in Transaktionen pro Minute (tpmC) von New-Order Transaktionen und der Preis in Dollar pro tpmC.*
- *TPC-D misst OLAP-Arbeitslasten (Online Analytical Processing), bei denen die meisten Transaktionen aus vielen komplizierten und lesenden Zugriffen auf die Datenbank bestehen. Auf Grund der Komplexität werden hier die Transaktionen häufig hintereinander und nicht wie bei OLTP parallel ausgeführt. Die Transaktionen sollen so schnell wie möglich ausgeführt werden. Der Benchmark beschreibt also analytische Datenbankapplikationen. (Strategiebildung für Aktienkäufe/-verkäufe, Analyse der Kreditfähigkeit von Kunden, Planung des Flugangebots einer Fluggesellschaft etc.). Leistungsmetriken sind die Query Processing Power, die angibt, wie schnell Anfragen bearbeitet werden, der Durchsatz an Anfragen pro Stunde und der Preis in Abhängigkeit der Leistung. Alle Metriken werden in Abhängigkeit der Datenbankgrösse angegeben.*

c) Sie wollen ein Datenbanksystem für Ihre Datenbankanwendung kaufen. Ihre Anwendung besteht sowohl aus Online Transaction Processing (OLTP), (d.h. es gibt viele Transaktionen, die die Datenbank modifizieren) als auch aus Analyse (d.h es gibt mehrere Transaktionen, die rein lesend sind und komplexere Anfragen an die Datenbank stellen). Das Verhältnis von modifizierenden Transaktionen und komplexen, reinen Lesetransaktionen beträgt etwa 80:20. Ihre Durchsatzanforderungen betragen etwa 14.000 Transaktionen pro Minute. Dabei sind die modifizierenden Transaktionen sehr zeitkritisch und sollten so schnell wie möglich ausgeführt werden, von den lesenden Transaktionen sind einige zeitkritisch, die meisten fordern jedoch keine schnellen Antwortzeiten.

Sie haben folgende Information aus TPC-C Benchmark Ergebnissen von zwei Datenbanksystemen DB1 und DB2. Die Tabelle 1 gibt den MQTh und die Presidaten für die beiden Systeme. (Erinnerung: MQTh = Maximum Qualified Throughput pro Minute und gibt nur den Durchsatz an New-Order Transaktionen an. Das ist ca. 45% des Gesamtumsatzes der TPC-C Arbeitslast).

Diagramme 1 bis 4 zeigen die Antwortzeitverteilungen der New-Order Transaktion (der wichtigsten modifizierenden Transaktion des Benchmarks) und der Stock-Level Transaktion (der längsten Lesetransaktion des Benchmarks) beim MQTh für die beiden Datenbanksysteme

Die Diagramme 5 und 6 zeigen die Antwortzeiten in Abhängigkeit vom Durchsatz für die New-Order Transaktion für die beiden Datenbanksysteme.

Datenbanksystem	MQTh in tpmC	Preis in $/tpmC
DB1	8.000	50
DB2	10.000	100

Tabelle 1: Durchsatz und Preisleistung von DB1 und DB2

Diskutieren Sie die Vorteile und Nachteile der beiden Datenbanksysteme im allgemeinen und speziell auf Ihre Anwendung bezogen. Benutzen Sie bei Ihrer Argumentation die Vorgaben Ihrer Anwendung und die Daten aus der Tabelle und den Diagrammen. (15 Punkte)

Lösung:

Ein schrittweiser Lösungsweg beinhaltet die Analyse der Applikation und wie die einzelnen Anforderungen der Applikation von den Datenbanksystemen erfüllt werden können.

- *Die Durchsatzanforderung von 14000 Transaktionen pro Minute wird von beiden Datenbanksystemen erfüllt, da die Angaben aus der Tabelle ja nur die New-Order Transaktionen beschreiben, die etwa 45% der Last ausmachen. DB2 hat eine höhere Durchsatzleistung und hat somit den Vorteil, dass es bei evtl. auftretender Erhöhung der Anforderungen länger genügend Kapazitäten bereitstellt.*

- *Hauptarbeitslast sind kurze, modifizierende Transaktionen, die sehr zeitkritisch sind. Beide Datenbanksysteme verhalten sich bei hoher Last bzgl. modifizerender Transaktionen identisch (Diagramme 1+3). Wird das Datenbanksystem aber nicht gesamt ausgelastet (was bei 14000 TA/min der Fall ist), verkürzen sich die Antwortzeiten bei DB2 sehr stark, bei DB1 jedoch nicht (Diagramme 5+6). Das ergibt einen Vorteil für DB2.*
- *Bei komplizierteren Anfragen liegen die Antwortzeiten bei DB1 wesentlich niedriger als DB2 (Diagramme 2+4). Das ergibt einen Vorteil für DB1. Jedoch ist diese Arbeitslast in der Applikation nicht so zeitkritisch.*
- *Die Kosten von DB2 sind mehr als doppelt so hoch wie die von DB1.*

Nun fehlt noch eine abschliessende Beurteilung. Die hängt davon ab, wie die einzelnen Punkte gewichtet werden. Ich tendiere zu DB2.

Aufgabe 4: TP-Monitore

a) Betrachten Sie das folgende Szenario einer (kleinen) Bank. Es werden drei Informationssysteme verwendet. Die Kundendatenbank KDB speichert Adressen von Kunden sowie Informationen über die pro Kunde existierenden Konti und Depots. Ausserdem sind dort die Konti selbst gespeichert. Die Depodatenbank DDB enthält die Aktienbestände jedes Kunden. Das Kursinformationssystem KIS schliesslich ist an die elektronische Börse angeschlossen und liefert die aktuellen Aktienkurse.

Sie sollen die drei Systeme mit Hilfe eines TP-Monitors verbinden. Dabei sind unter anderem die folgenden Operationen zu unterstützen:

(a) Ein Kunde kauft eine bestimmte Menge Aktien (Parameter: Kundennummer, Aktiennummer, Anzahl). Der Kaufpreis wird automatisch seinem Konto belastet, und die Aktien ins Depot eingefügt. (Nehmen Sie der Einfachheit halber an, dass jeder Kunde nur ein Konto besitzt.)

(b) Die Sektionsleiterin für Börsengeschäfte möchte den durchschnittlichen Wert aller Kundendepots wissen.

Skizzieren Sie die Struktur eines solchen Systems. Beschreiben Sie insbesondere, welche Applikationsserver Sie benötigen, welche Services diese (mindestens) anbieten müssen, und wie die beiden beschriebenen Operationen umgesetzt werden. (10 Punkte)

Lösung:

TP-Monitor-Anwendungen verwenden eine Drei-Schichten-Architektur, in der Applikationsserver zwischen Clients und Datenbanken liegen. In diesem konkrekten Beispiel verwenden wir drei Applikationsserver, die jeweils eine oder mehrere Datenbanken kapseln.

Der Customer-*Server kapselt die KDB-Datenbank und implementiert (mindestens) die Services debit (Kundennummer, Betrag), credit (Kundennummer, Betrag) und getDepot (Kundennummer). Der* Depots-*Server kapselt* *sowohl die DDB-Datenbank als auch die KIS-Datenbank und bietet die Services addStocks (Depotnummer, Aktiennummer, Anzahl) und averageValue() an. Letzterer berechnet den Durchschnittswert aller Depots. Der* Rates-*Server kapselt die KIS-Datenbank und bietet den Service currentRate (Aktiennummer) an.*

Operation (a) *(Aktienkauf) läuft folgendermassen ab: Das Clientprogramm, das auf dem Rechner der KundenbetreuerIn läuft, ermittelt mit Hilfe von currentRate() den momentanen Kurs der gewünschten Aktie und berechnet den Gesamtwert der Transaktion. Dieser Wert wird mittels debit() vom Konto des Kunden abgezogen. Dann wird mittels getDepot() seine Depotnummer ermittelt und mittels addStocks() die gewünschte Anzahl Aktien eingefügt.*

Operation (b) *(Durchschnittswert) wird vollständig vom* Depots-*Server berechnet. Der Service averageStocks() ermittelt zu jeder Aktie in jedem Kundendepot den aktuellen Kurs (durch Anfragen bei der DDB- bzw. KIS-Datenbank), berechnet den Wert jedes Depots und dann den Durchschnittswert aller Depots, der schliesslich zurückgeliefert wird.*

Andere Strukturierungen des Systems sind natürlich auch möglich und können unter Umständen sinnvoll sein. So wäre es z.B. denkbar, einen einzigen Applikationsserver zu installieren, der auf alle Datenbanken Zugriff hat und alle Dienste anbietet. Vom Gesichtspunkt der Programmierung her ist es hingegen am einfachsten, jeden Applikationsserver an genau eine Datenbank zu binden, da dann im Programmcode der Server nicht zwischen den verschiedenen Datenbanken unterschieden werden muss. Eine weitere Option besteht darin, für Dienste, die sehr häufig benötigt werden, spezielle Server zu definieren, von denen dann mehrere auf verschiedenen Rechnern gestartet werden. Auf diese Weise lässt sich die Last des Systems im Netzwerk verteilen. In unserer Beispielanwendung würde es z.B. Sinn machen, den addStocks-Service des Depots-Servers vom averageValue-Service zu trennen. Dadurch wird vermieden, dass die (relativ kurzen) AddStock-Transaktionen von den lang laufenden AverageValue-Transaktionen blockiert werden.

b) Geschachtelte Transaktionen, wie sie von dem TP-Monitor-System Encina angeboten werden, können in bestimmten Fällen zu Problemen führen. Beschreiben sie kurz die Problematik und mögliche Lösungsansätze. (15 Punkte)

Lösung: Der Kern des Problems besteht darin, dass Encina zwar geschachtelte Transaktionen anbietet, die meisten Datenbanken aber nicht. Der XA-Standard, der die Kommunikation zwischen TP-Monitor und Datenbank spezifiziert, kennt nur flache Transaktionen. Das Problem tritt auf, wenn Encina zwei Subtransaktionen derselben Elterntransaktion an die gleiche Datenbank schickt. Das erwartete Verhalten im Sinne des geschachtelten Transaktionsmodells wäre, daß die beiden Transaktionen isoliert voneinander ablaufen, daß ihre Sperren am Ende jedoch an die Elterntransaktion weitergereicht werden. Ergebnisse werden erst commited, wenn die Elterntransaktion beendet ist. Diese Verhalten ist jedoch nicht möglich, wenn die Datenbank keine Subtransaktionen kennt.

Lösungsansätze: Encina bietet mehrere Strategien zur Abbildung von geschachtelten Transaktionen auf flache Transaktionen des Datenbanksystemes an. Je nach gewünschter Semantik im Einzelfall kann die geeignetste Strategie gewählt werden. Die verschiedenen Strategien finden sich im "Encina Toolkit Server Core Programmer's Reference", Kapitel 28.5. Grundsätzlich gibt es die Möglichkeit, jede Subtransaktionen auf eine eigene DBMS-Transaktion abzubilden oder alle Subtransaktionen als eine *DBMS-Transaktion auszuführen. Im ersten Fall ist die Isolation zwischen den Subtransaktionen gewährleistet, Sperren können jedoch nicht an die Elterntransaktion zurückgegeben werden. Zudem commited jede Subtransaktion unabhängig. Im zweiten Fall sind Sperren einer Subtransaktion für die Elterntransaktion zugreifbar, es besteht jedoch keine Isolation zwischen Subtransaktionen (die meisten DBMS werden solche Subtransaktionen allerdings garnicht parallel ausführen können), und das Zurücksetzen einzelner Subtransaktionen ist auch nicht möglich. Weitere Abbildungsmechanismen werden von Encina definiert, aber von den meisten DBMS nicht unterstützt.*

Die geeignete Strategie hängt also stark von der jeweiligen Anwendung ab. Wurden Subtransaktionen lediglich als Strukturierungsmechanismus für die Anwendungsprogramme verwendet, so ist die Ausführung als eine grosse Transaktion sinnvoll. Wurden Subtransaktionen bewußt gewählt, um durch Parallelausführung die Ausführung zu beschleunigen oder um Fehlertoleranz durch gezieltes Zurücksetzen einzelner Subtransaktionen zu erreichen, so sollte die Strategie der Ausführung jeder Subtransaktion als separate DBMS-Transaktion gewählt werden. In jedem Fall muss sich die ProgrammiererIn der speziellen Semantik bewusst sein, da z.B. bei ungeschickter Programmierung sehr leicht Deadlocks entstehen können.