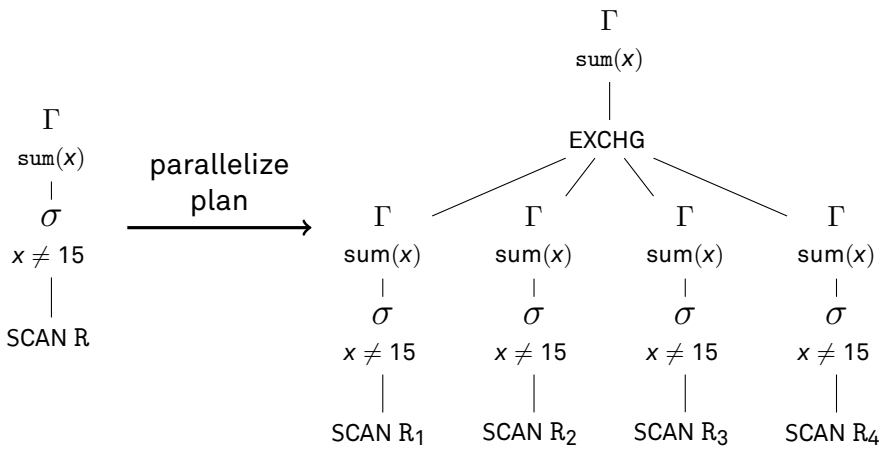
# 4. Übungsblatt

Ausgabe: 04. November 2019 · Besprechung: 18. November 2019

## Einleitung

Um Systemresourcen wie CPU-Rechenkerne und Hauptspeicherbandbreite effektiv zu nutzen müssen Programme parallelisiert werden. Auf diesem Übungsblatt geht es um die Parallelisierung des experimentier-Datenbanksystems `WeeDB` mit Exchange-Operatoren.

Exchange-Operatoren sind eine klassische Parallelisierungstechnik für Datenbanksysteme. Sie werden wie andere relationale Operatoren in den Anfrageplan integriert und steuern die parallele Ausführung und Kommunikation. Ein Vorteil von Exchange-Operators ist, dass die übrigen Verarbeitungsschritte weitgehend unverändert bleiben. Der folgende Plan wird beispielsweise mit einem N:1 (Beispiel N=4) Exchange-Operator parallelisiert:




Der Exchange Operator started vier Threads die jeweils einen Subplan $\text{SCAN} \to \sigma \to \Gamma$ ausführen. Jeder der Threads bearbeitet eine Partition $R_1, \cdots, R_4$ von $R$. Danach konkateniert der Exchange-Operator die Teilergebnisse in eine zusammenhängende Relation. Der Aggregationsoperator, der die Wurzel des gegebenen Plans darstellt, aggregiert die Tupel der Relation zum Anfragergebnis auf.

## Aufgabe — Planumformung

Um `WeeDB` mit paralleler Anfrageauswertung zu erweitern soll zunächst die Planumformung von einem sequentiellen Plan zu einem parallelen Plan implementiert werden. Dazu erweitern Sie die Basisklasse `RelOperator` in `BaseOperator.h` um die Methode

```
virtual std::vector < RelOperator* > parallelize ( int numThreads );
```

Die Implementierungen der Methode können dann je Operator in `Operators.h` vorgenommen werden. Die Implementierung von `parallelize(...)` ruft zunächst `parallelize(...)` auf dem Kindoperator auf. Dieser Aufruf liefert 1 bis `numThreads` parallele Instanzen des Kindoperators. An jede Kindinstanz wird dann eine neue Instanz des Operators selbst angefügt (vgl. Planumformung oben).

Manche Operatoren wie Aggregation erfordern Synchronisation, z.B. ist die Summe über eine Relation erst verfügbar wenn alle Teilsummen addiert wurden. Deshalb können Operatoren in der `parallelize(..)` Methode veranlassen, dass Teilergebnisse zusammengeführt werden indem sie Exchange-Operators platzieren. Betrachten Sie als Beispiel die folgende Implementation der Methode für den Aggregationsoperator:

```
// parallelize der Klasse AggregationOp
std::vector<RelOperator*> parallelize (int numThreads) {
    std::vector<RelOperator*> result;
    // Rekursiver Aufruf
    std::vector<RelOperator*> children = child->parallelize(numThreads);
    // Füge je parallele Kind-Instanz einen Aggregationsoperator an
    std::vector<RelOperator*> aggregations;
    for (auto c : children) {
        aggregations.push_back(new AggregationOp(type, c));
    }
    // N:1 Exchange (parallele Ausführung, konkateniere Teilergebnisse)
    ExchangeOp* xchg = new ExchangeOp(aggregations);
    // Aggregiere Teilergebnisse
    result.push_back(new AggregationOp(ReduceType::SUM, xchg));
    delete this;
    return result;
}
```

**Hinweise:**
Nach einem Exchange-Operator wird die Ausführung sequentiell fortgesetzt. Berücksichtigen Sie, dass Sie gegebenenfalls einen Exchange-Operator als neue Wurzel des Plans platzieren müssen, siehe `PlanParallelizer::apply()` in `Operators.h` (auskommentiert). Der SCAN-Operator benötigt keinen rekursiven Aufruf, da er keine Kinder hat. Hier unterteilt `parallelize(...)` die Relation auf mehrere SCAN-Operatoren.

## Aufgabe — Implementierung des Exchange Operators

Implementieren Sie den Exchange-Operator für die drei Verarbeitungsmodelle. Der Exchange-Operator erbt von der Klasse `RelOperator` aus `BaseOperator.h` und soll somit die Interfaces für jedes Verarbeitungsmodell implementieren. Als Beispiel erläutern wir hier die Implementierung für das Volcano-Verarbeitungsmodell. Die Exchange-Operatoren haben im wesentlichen zwei Aufgaben:

1. Sie veranlassen die parallele Ausführung von Teilplänen.
2. Sie stellen Puffer bereit und führen Puffer zusammen.

Die `open()`-Methode alloziert zunächst Speicherplatz in je einem Puffer für die Ergebnisse der Teilpläne. Danach wird die parallele Ausführung jedes Teilplan gestartet. Dazu werden mit `pthread` oder `std::thread` Threads gestartet, die die `PullDriver::volcano(...)`-Methode aus `Operators.h` ausführen, z.B.

```
// Starte Threads; deg: Anzahl Kinder des Exchange Operators
std::vector < std::thread > t(deg);
for ( int i=0; i<deg; i++) {
    t[i] = std::thread(&PullDriver::volcano, children[i], &buffers[i]);
}
```

In der open()-Methode muss außerdem sicher gestellt werden, dass die Puffer hinreichend befüllt sind, sodass der Pufferinhalt im Anschluss konsumiert werden kann. Dazu können Sie zum Beispiel folgendermaßen auf die Beendigung der Threads warten:

```
// Warte auf Fertigstellung der Threads
for ( int i=0; i<deg; i++) {
    t[i].join();
}
```

Die `next()`-Methode hat danach nur noch die Aufgabe die Pufferinhalte Element für Element zurück zu geben. Im Anschluss gibt die `close()`-Methode die Puffer frei.

**Hinweise:**
Implementieren Sie zunächst den Exchange-Operator für parallele Volcano-Verarbeitung mit Hilfe der Beschreibung. Implementieren Sie danach den Exchange-Operator für Operator-at-a-time und Vector-at-a-time-Verarbeitung.

## Aufgabe — Analyse

Erweitern Sie die Analyse von Blatt 3 um den Aspekt der Parallelität. Welche Performance-Steigerung können Sie erreichen, wenn die den Grad der Parallelität erhöhen? Wie hoch ist der erreichte Durchsatz und die Auslastung der Speicherbandbreite?