# Klausurinhalte SDMS

## Contents

Single-Node DBMS ... 4
Layer ... 4
Pages ... 4
Slotted Pages ... 4
Tuple Pointers (Record IDs) ... 4
Row und Column Store ... 5
Buffer Manager ... 5
Eviction Strategies ... 5
Clock Replacement (Annäherung an LRU) ... 5
Access Methods ... 6
B-Trees ... 6
Basic B-Tree ... 6
B+ Tree ... 7
Distributed DBMS ... 7
Shared-Nothing Architektur (Traditionell) ... 8
Shared-Data Architektur (Cloud) ... 8
Datenverteilung ... 8
Fragmentierungsschemas (Partitioning) ... 9
OLAP (Online Analytical Processing) ... 10
Partitionsschema ... 11
OLAP Query Verarbeitung ... 11
Indices ... 11
OLAP Query Execution ... 12
OLAP-Architektur ... 12
OLTP (Online Transaction Processing) ... 13
Partitionsschema ... 13
Distributed OLTP ... 13
OLTP-Architekturen ... 14
Non-partitioned, replicated ... 14
Non-partitioned, shared Data ... 15
Replikation für Fehlertoleranz ... 15
Distributed OLTP Processing Model ... 16
Commit Protokolle ... 16

Cloud OLTP ... 17
Multi-Writer OLTP Databases in the Cloud ... 18
Query Optimierung ... 18
Cardinality Estimation & DeepDB ... 19
RSPN ... 20
Distributed Optimierung ... 21
Query Plan ... 21
Execution Models ... 22
Physische Operatoren ... 22
Kosten ... 23
Cloud ... 23
Concurrency Control ... 24
Serializability ... 24
CC-Schemata ... 25
Lock-based Concurrency Control ... 25
Deadlocks ... 26
Multiple Granularity Locking ... 26
Multi-Versoning Concurrency Control ... 27
Snapshot Isolation ... 27
Snowflake ... 27
Snwoflake Table Structure ... 28
Pruning ... 28
Indexing ... 29
Search Optimization Service ... 29
Materialization und Reuse ... 30
Automatisches Clustering ... 30
Clustering Metriken ... 30
MapReduce ... 31
Apache Hadoop ... 31
Apache Spark ... 32
Streaming ... 32
Stream ... 33
Stream Processing ... 33
Stream Windows ... 33
Apache Flink ... 34
Security und Privacy ... 36

Security Toolbox ..... 37
Merkle Trees ..... 37
Secure Hardware ..... 38

# Single-Node DBMS

Ein Single-Node DBMS ist ein Datenverwaltungssystem, welches auf einem einzigen Computer (Node) oder Server ausgeführt wird. Es hat mehrere Layer und Komponenten, welche die Daten verwalten, verarbeiten und speichern.

## Layer

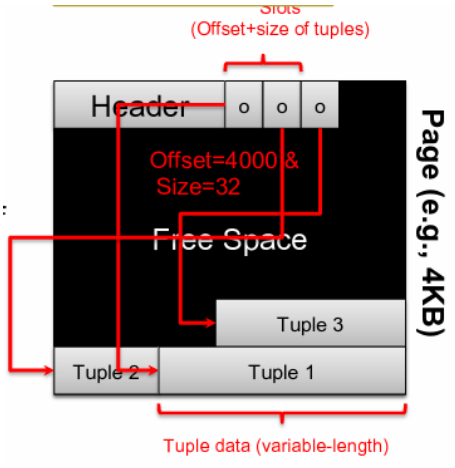
1. SQL Processing: Der Anfangspunkt des DBMS an dem SQL Queries angenommen werden. Das System verarbeitet diese Anfragen, um die angeforderten Operationen zu verstehen.
2. Query Optimization: Das DBMS optimiert die SQL Queries um sicherzustellen, dass diese möglichst effizient ausgeführt werden. Hier wird der beste Ablaufplan aufgrund einer Kostenfunktion ausgewählt.
3. Query Execution: Diese Komponente führt vorher gewählte Query aus.
4. Access Methods: Dieser Layer beinhaltet verschiedene Methoden, um auf den Datenspeicher zuzugreifen.
5. Buffer Pool: Der Buffer Pool ist ein in-memory cache, welcher häufig verwendete Pages beinhaltet, dies reduziert Zugriffszeiten (cache hit, cache miss).
6. Disk Manager: Diese Komponente ist für die Verwaltung des physischen Speichers verantwortlich. Er kommuniziert mit dem Speicher, um Pages zu schreiben oder zu lesen, verwaltet, wo diese gespeichert werden, und sorgt für Datenintegrität und -beständigkeit
7. Database Storage: Der niedrigste Layer des DBMS repräsentiert den physischen Speicher, dieser Layer ist in Blöcke oder Pages unterteilt, welche die Basic Unit des Datenspeichers darstellen.

## Pages

Eine Page ist eine fundamentale Einheit von Datenspeicher in einem DBMS. Es hat eine fest gesetzte Größe, welche typsicherweise ein paar Kilobyte groß sind. Pages werden genutzt um Tuple (Zeilen einer Tabelle) zu speichern. Das DBMS liest und schreibt Daten als Pages auf den Speicher um Datenzugriffe zu reduzieren. Pages haben eine Header in dem Page-Size und Slot-Size etc gespeichert sind.

## Slotted Pages

Eine slotted Page ist eine spezielle Struktur, die innerhalb einer Page verwendet wird um Tuple zu speichern. Jeder der Slot beinhaltet den Offset zur Tuple-Data und die größe. Die Tuple werden dann Back-To-Front gespeichert.




Fixed length Tuple werden direkt am Beginn der Page gespeichert.

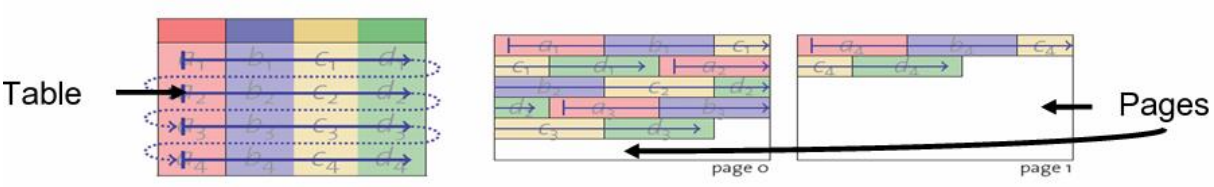
- Insert Operation: Tuple am Ende der Page und Pointer hinzufügen
- Read Operation: Erst Pointer lesen und dann Daten holen
- Delete Operation: Binary Flag pro Slot setzen ob er benutzt wird

## Tuple Pointers (Record IDs)

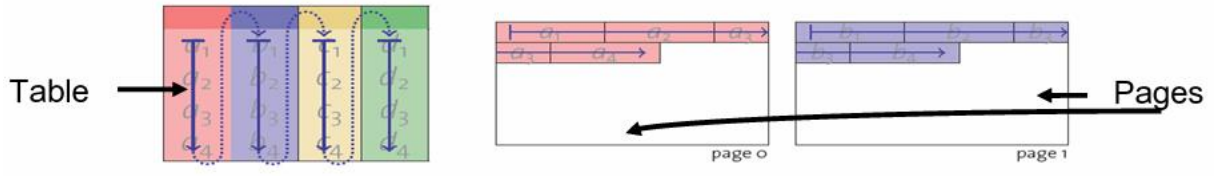
Record IDs sind Pointer zu einem Tuple und setzen sich aus der Eindeutigen Page-ID und der Slot Nummer zusammen. Erst im Page-Table schauen ob die Daten noch im Buffer Pool sind oder ob man die Daten aus dem Speicher holen muss.

### Row und Column Store

Bei einem Row-Store werden die Attribute eines Tupels in einer Page gespeichert, diese Methode ist gut geeignet für Insert, Update und Delete.




Beim Column-Store werden die Attribute einer Spalte in einer Page gespeichert und nicht als Zeile, so können Daten besser komprimiert werden und Aggregate Funktionen (SUM; AVG; MAX, …) einfacher ausgeführt werden.




### Buffer Manager

Der **Buffer Manager** überwacht den Buffer Pool, entscheidet welche Pages ausgetauscht werden, falls der Buffer Pool voll ist.

**Buffer Pool**: Der Buffer Pool ist ein in-Memory Speicher, welcher häufig genutzte Pages speichert, der Buffer Pool hat allerdings nur eine begrenzte Anzahl an Frames, jedes dieser Frames kann eine Page halten.

**Page Table**: Der Page Table speichert einen Pointer zu den Daten im Buffer Pool und weitere Metadaten wie ein Dirty Flag und ein Reference Counter.

Der Buffer Manger verwendet hierfür vorallem die Methoden Pin (Page aus dem Pool anfragen) und Unpin (Page freigeben)

- **Pin:** Prüfen, ob Page im Buffer, falls nein und Buffer voll -> Evicten, Page laden, pincount +1
- **Unpin:** pincount -1, falls dirty, dirtBit auf 1 setzen

## Eviction Strategies

Wenn eine neue Page angefordert wird und der Buffer Pool voll ist, muss der Buffer Manager ein Frame wählen, welches geleert wird. Dabei ist wichtig, dass möglichst Frames die häufig verwendet werden, im Buffer Pool bleiben.

- **Random:** Eine Zufällige Page mit pinCount=0 (bedeutet, sie wird gerade nicht verwendet) entfernen.
- **FIFO** (First-In-First-Out): Page entfernen in der Reihenfolge, in der sie in den Buffer geladen wurde, durch ein einfaches Queue System
- **LRU** (Least-Recently-Used): Die Page entfernen, welche als längstes nicht mehr verwendet wurde. Wenn man die PageIDs nach Zugriffszeit speichert, führt es allerdings zu einem großen Overhead.

### Clock Replacement (Annäherung an LRU)

Der Clock Algorithmus hat einen deutlich geringeren Overhead als LRU, kommt diesem aber sehr nah.

- Kreisgeschlossener Buffer: Die Frames sind in einem „Kreis“ angeordnet
- Reference Bit: Jede Page hat ein Reference Bit, wenn auf die Page zugegriffen wird, wird dieses Bit auf 1 gesetzt
- Clock Hand: Ein Pointer, der durch die Frames läuft und anzeigt, welches Frame als nächstes für die Eviction betrachtet wird.
- Eviction Process:
    - Der Algorithmus checkt die Aktuelle Clock Hand Position
    - Wenn das Reference Bit auf 1 steht, wird es auf 0 gesetzt und die Hand bewegt sich einen Slot weiter
    - Wenn das Reference Bit auf 0 steht, wird die Page entfernt (falls sie nicht gepinnt ist, dann wird refBit wieder auf 1 gesetzt)

# Access Methods

Zugriffsmethoden sind Techniken, um Daten per spezifischen Suchkriterien zu finden.

- Key-Lookups/ Single-Value Searches: Daten werden mit dem Gleichheitsprädikat gesucht, es werden Daten mit einem bestimmten Wert gesucht
    - SELECT * FROM CUSTOMER WHERE cid=7
- Range Searches: Daten werden in einem bestimmten Wertebereich zurückgegeben
    - SELECT * FROM CUSTOMER WHERE age>=20 and age<=30
- Pattern Searches: Daten mithilfe von bestimmten String Pattern finden
    - SELECT * FROM CUSTOMER WHERE last_name LIKE 'a%'

# B-Trees

B-Bäume sind ein Baumbasierter Index (geeignet für Key und Range Lookups), es gibt auch noch Hash-basierte (nur für Key-Lookups) und andere Methoden. B-Trees sind im Prinzip Binär-Bäume, von diesen gibt es auch unterschiedliche Varianten.

## Basic B-Tree

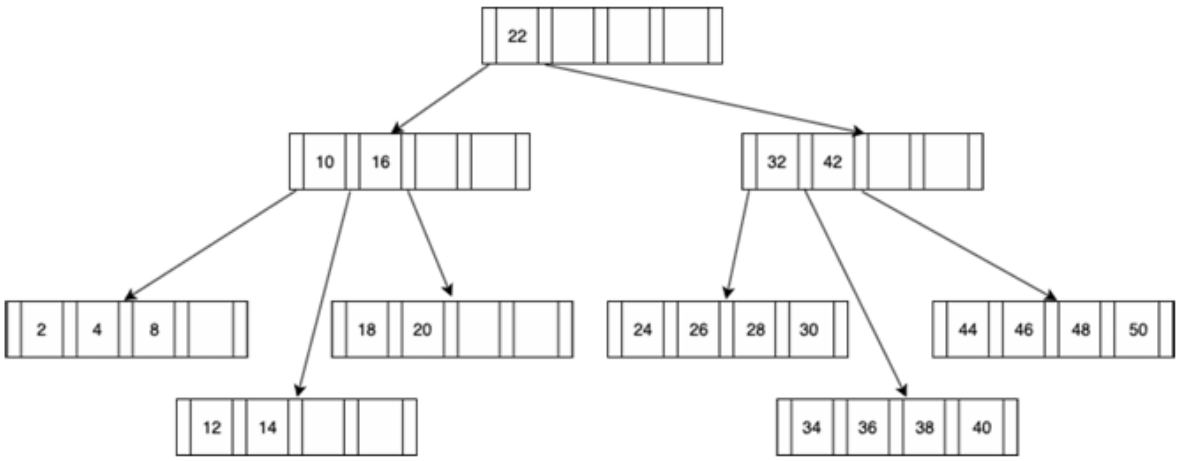
Jeder Index-Node speichert ein Index Key und einen Pointer zu einem Tupel, jeder Index Node hat >2 Kinder und ist immer balanced.

Nodes haben einen minimalen Füllgrad **M** und einen maximalen Füllgrad **2M**, der Füllgrad legt fest, wie viele Keys in einem Node gespeichert werden können. Root Node kann <M Einträge haben.

Jedes innenliegende Node und Blatt ist als extra Datenbank Page gespeichert. Die Pointers zu den Child Nodes sind mit Page-IDs gespeichert und die Pointer zu den Tupel mit den RIDs.

- Lookup: Gibt Daten für einen Key oder eine Keyrange zurück.
- Insert, ein neuer Key/Pointe (RID) zu einem Tuple wird dem index hinzugefügt
    - Falls der Node bereits voll ist, wird der neue Tupel angehängt, der Tupel dann in der Mitte geteilt und der mittlere Index geht in den Parent Node an die korrekte Stelle.
- Update: Kann wie Insert oder Delete geschehen
- Delete: Vorhandener Key /Pointer (RID) zu einem Tupel wird aus dem Index entfernt
    - Falls Leaf weniger als halb voll (also <M) ist, gibt es unterschiedliche Strategien mit dem zu wenig befüllten Leaf umzugehen
        - Gelöschter Key wird durch den Predicessor ersetzt
        - Gelöschter Key wird durch den Successor ersetzt
            - Hier kann nun entweder ein anderer Wert rotiert werden, falls das andere leaf nun zu wenige Einträge hat

- Oder die Leafs können zusammengeführt werden




### B+ Tree

Innere Nodes speichern nur Separatoren (surrogate keys) und keine Pointer zu Tupeln. Ale Keys zu den Tuepln werden auf dem Leaf-Level gespeichert. Die Leafs sind als double-linked List verbunden, damit werden Range searches ermöglicht

Vorteile:

- Da keine RIDs gespeichert werden müssen, können die Keys mehr in der Breite verteilt werden (Fan-Out), das sorgt für eine geringere Tiefe und somit schnelleren I/O Zugriffen.
- Da die Keys sequentiell alle auf einer Ebene gespeichert werden, macht es Range suchen sehr effizient

Methoden

- **Search**
  - Key Lookup: Zum Leaf und Tupel zurückgeben
  - Range Search: Zum Leaf mit der unteren Grenze und dann von links nach rechts zur oberen Grenze
- **Insert:** Immer auf Leaf level einfügen, Splitten eines Nodes braucht einen neuen Seperator im Parent Node
- **Delete:** Immer auf Leaf Level, Reorganisieren wie im B-Tree

## Distributed DBMS

DBMS können entweder innerhalb eines Datencenters verteilt sein (Parallel DBMS) oder über mehrere Datencenter verteilt sein (Geo-distributed DBMS).

Parallel DBMS können entweder homogen sein (jede Node in einem Cluster von Nodes führt dieselbe DBMS Software aus) oder zentralisiert (Ein festgelegter Koordinator-Node welcher alle Queries kompiliert und optimiert, danach führen alle Nodes diese Query parallel aus.

**Intra-Query Parallelismus**: Eine Query wird parallel auf mehreren Maschinen in einem Cluster ausgeführt.

**Inter-Query Parallelismus**: Verschiedene Queries werden parallel auf mehreren Maschinen in einem Cluster ausgeführt.

- Speedup: Performanz für eine feste Datengröße und wachsende Ressourcen
  - $Speedup = \frac{small\ system\ elapsed\ time}{large\ system\ elapsed\ time}$
- Scaleup: Performanz für wachsende Daten und Ressourcen
  - $Scaleup = \frac{small\ system\ small\ problem\ elapsed\ time}{big\ system\ big\ problem\ elapsed\ time}$

## Shared-Nothing Architektur (Traditionell)

**Datenverteilung:** In einer shared-nothing Architektur hat jeder Node des verteilten Systems sein eigenen Datenanteil, die Daten sind über die verschiedenen Nodes verteilt und jeder Node arbeitet auf seinem Anteil.

**Processing:** Jeder Node verarbeitet die Queries auf seinen privaten Daten, ohne von den Ressourcen oder Daten der anderen Nodes abhängig zu sein. Dadurch wird der Datentransfer zwischen den Nodes und somit der Netzwerk overhead reduziert.

**Skalierbarkeit:** Die shared-nothing Architektur skaliert gut horizontal durch Hinzufügen neuer Nodes, jedes mit seinen eigenen Daten und Ressourcen.

## Shared-Data Architektur (Cloud)

**Datenverteilung:** Mehrere Server haben Zugriff auf dieselben Daten per Netzwerk.

**Processing:** Rechenserver können die Queries auf den gesamten Daten ausführen, das macht das System deutlich flexibler und ist praktisch für häufige Datenzugriffe auf unterschiedliche Teile der Daten.

**Skalierbarkeit:** Die Rechenserver können dynamisch auf unterschiedliche Teile der Daten angepasst werden.

| Shared Data | Shared Nothing |
|---|---|
| Gute Lastverteilung | Geringe Latenz |
| Fehlertoleranz | Einfach zu implementieren |
| Hohe Latenz | Last-Inbalance und Node ausfälle |

## Datenverteilung

Daten werden in zwei Schritten verteilt. Parallele DBMS verwenden meistens horizontale Fragmentierung mit einem Fragment pro Server, häufig gibt es mehr Fragmente als Server.

1. Fragmentierung, Tabelle wird in kleinere Fragmente aufgeteilt, dies werden shards oder fragments genannt
   a. Daten können zeilenweise (Horizontal) oder spaltenweise (Vertikal) geteilt werden
2. Allokation, Entscheiden, welche Fragmente zu welchem Server geleitet werden

Eigenschaften:

- **Co-Partitioning**: Co-Partitionierung bezieht sich auf die Praxis, mehrere Tabellen nach dem gleichen Partitionierungsschema zu partitionieren, idealerweise basierend auf den Join-Schlüsseln. Dadurch wird sichergestellt, dass verwandte Daten aus verschiedenen Tabellen in derselben Partition landen, was die Notwendigkeit für kostspielige Datenübertragungen über das Netzwerk während Join-Operationen minimiert.
- **Pruning von Partitionen**: Partition Pruning ist der Prozess des Überspringens unnötiger Partitionen während der Ausführung einer Abfrage. Wenn das Partitionierungsschema gut

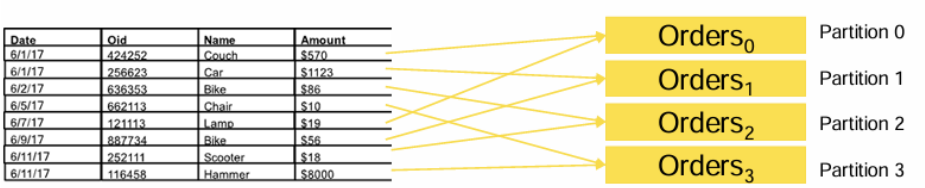
gestaltet ist, kann das System schnell bestimmen, welche Partitionen für eine Abfrage relevant sind und nur diese scannen, was die Abfrageleistung erheblich verbessert.

- **Datenverteilungs-Schieflagen (Skew)**: Eine gut durchdachte Partitionierungsstrategie kann dabei helfen, Situationen zu verhindern, in denen ein Knoten einen deutlich größeren Anteil der Daten als andere erhält, was als Datenverteilungs-Schieflage bekannt ist. Die Vermeidung von Schieflagen ist entscheidend, um sicherzustellen, dass alle Knoten im System ungefähr die gleiche Menge an Arbeit verrichten, was wiederum die Gesamteffizienz und -leistung des Systems maximiert.

## Fragmentierungsschemas (Partitioning)

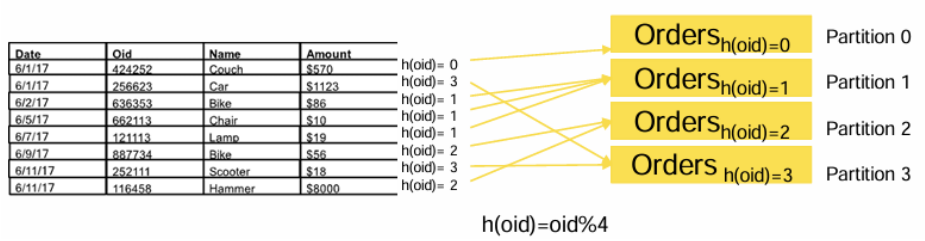
Round-Robin: Tupleweise Zuweisung zu den einzelnen Partitionen

- Pro: Verhindert Datenverteilungs Skew
- Con: Pruning ist nicht möglich



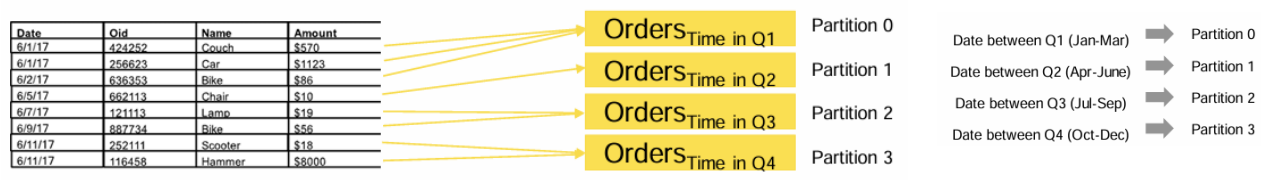

Hash: Verwendet Hash Funktion, um Tupel zuzuweisen

- Pro: Pruning für key-Lookups und Range Queries
- Con: Anfällig für Datenverteilungs Skew




Range: definiert Range-Partitionen nach bestimmten Regeln

- Pro: Pruning für key-Lookpups, verhindert Skew
- Con: Kein Pruning für Range Queries




- Predicate-based Reference Partitioning: Bei der predikat-basierten Referenzpartitionierung (PREF) wird minimale Datenredundanz auf der Tupel-Ebene eingeführt, um die Datenlokalität zu maximieren. Ein Beispiel für PREF wäre, Tabelle R durch S mit dem Prädikat R.B = S.B zu partitionieren.
- Automated Partitioning: Beim automatisierten Partitionierungsdesign mit PREF wird für jede Tabelle, gegebenenfalls basierend auf einer Arbeitslast von SQL-Abfragen, ein Partitionierungsschema (HASH oder PREF) zugewiesen. Das Ziel ist es, die Datenlokalität für verteilte Joins zu maximieren und die Datenredundanz zu minimieren.

- Schema Partitioning: Beim schema-getriebenen (Schema-driven, SD) Partitionierungsdesign repräsentieren Fremdschlüssel im Schema die Join-Pfade. Es gibt auch einen arbeitslast-getriebenen (Workload-driven, WD) Ansatz, bei dem die Arbeitslast verwendet wird, um die Join-Pfade abzuleiten. Beim Schema-Driven Partitioning Design wird ein Schema-Graph verwendet, um die Partitionierungsstrategie zu bestimmen.

## OLAP (Online Analytical Processing)

OLAP-Systeme sind darauf ausgelegt, komplexe Analysen und ad-hoc queries auszuführen, anstelle von Transaktionen. Diese Systeme sind vor allem für Aggregatsfunktionen optimiert. OLAP-Datenbanken speichern historische Daten in einem mehrdimensionalen Schema.

**OLAP Cubes** verwenden ihr eigenes Datenmodell und Operationen zur Analyse von Daten. Daten sind logisch in einem mehrdimensionalen Array organisiert, ähnlich einer N-dimensionalen Excel-Tabelle. Jede Dimension beschreibt einen anderen Aspekt (z.B. Produkte, Geschäftsstandorte), und Fakten sind Messungen für die eigentliche Analyse (z.B. vom Kunden gezahlter Preis).

**Slice and Dice** bezieht sich auf das Filtern und Betrachten von Daten aus verschiedenen Perspektiven innerhalb eines OLAP-Cube. Ein "Slice" ist eine zweidimensionale Darstellung eines Cube, die durch Auswahl eines einzelnen Wertes für eine der Cubes Dimensionen entsteht, wobei "Dice" das Erstellen eines sub-Cubes durch Auswahl von mehr als einem Wert über mehrere Dimensionen hinweg bedeutet.

**Roll-Up** (Aggregation) und **Drill-Down** (Disaggregation) sind Techniken, um Daten auf unterschiedlichen Detailebenen zu analysieren. Beim Roll-Up werden Daten entlang einer oder mehrerer Dimensionen aggregiert, um eine höhere Abstraktionsebene zu erreichen. Beim Drill-Down wird der Prozess umgekehrt, um detailliertere Daten zu erhalten.

**OLAP-Abfrageverarbeitung** erfordert das Scannen und Verknüpfen potenziell großer Dimensions- und Faktentabellen. Datenbanksysteme haben mehrere Optimierungen entwickelt, darunter Cross-Join-Pläne, Semi-Join-Pläne sowie Speichermanagement-Techniken wie Bitmap-Indizes, Join-Indizes und materialisierte Ansichten.

1. Joins, um Dimensionstabellen mit der Faktentabelle zu verbinden
2. Filter auf Dimeonsonstabellen um einen Dice oder Slice der Daten zu erhalten
3. Group-By Dimension für Roll-Up oder Drill-Down
    a. Drill-Down: Neue Attribute zu Group By hinzufügen
    b. Roll-Up, Attribute aus Group-By entfernen
4. Aggregatfunktionen, um Fakten in Faktentabelle zusammenzufassen

**MOLAP** (Multi-dimensional OLAP) verwendet spezialisierte DBMS-Lösungen, die sich auf ein natives Datenmodell zur Speicherung und Verarbeitung von Datenwürfeln (multidimensionale Arrays) stützen. Beispiele für kommerzielle Produkte sind IBM Cognos Analytics.

Bei **ROLAP** (Relational OLAP) werden Würfeldaten in einer relationalen Datenbank unter Verwendung mehrerer Tabellen (Sternschema) gespeichert. Würfeloperationen werden als SQL-Abfragen formuliert. Beispiele für kommerzielle Produkte umfassen Cloud- oder On-Premise-DBMS wie Oracle, IBM DB2 und Cloud-DBMS wie Google BigQuery, Amazon Redshift und Snowflake.

**Dimensionstabellen** beschreiben die Daten einer Achse eines Würfels, wie z.B. Standortdaten (Stadt, Land usw.). Sie sind typischerweise nicht sehr groß und enthalten oft eine Hierarchie für Roll-Up und Drill-Down-Operationen, z.B. Region -> Land -> Stadt oder Jahr -> Monat -> Tag.

In **Faktentabellen** werden numerische Daten als Fakten für die Aggregation gespeichert (z.B. SUM(Preis)), sowie Fremdschlüssel zu Dimensionen (Produkt, Zeit usw.). Faktentabellen sind in der Regel groß und wachsen ständig. Sie enthalten zusätzliche Spalten für Fremdschlüssel zu allen Dimensionstabellen.

## Partitionsschema

OLAP verwendet ein Stern Schema

1. Fact Table und größter Table werden auf dem Join-Key co-partitioniert: Dies erleichtert effiziente Joins, da die Join-Operationen lokal auf jedem Server durchgeführt werden können, ohne dass Daten über das Netzwerk übertragen werden müssen.
2. Resultierende Partitionen werden auf Servern zugewiesen: Dies ermöglicht lokale Joins und reduziert die Notwendigkeit für umfangreiche Netzwerkkommunikation.
3. Alle anderen Dimensionstabellen werden auf allen Knoten repliziert: Dadurch können Joins mit diesen Dimensionstabellen ohne Netzwerkkommunikation durchgeführt werden, was die Abfrageleistung verbessert.

## OLAP Query Verarbeitung

OLAP Queries müssen potentiell große Dimensionen und Faktentabellen scannen und verbinden. Hierfür wurden mehrere Optimierungsmethoden entwickelt:

**Naive Join Processing Plan**: Diese Strategie führt Joins inkrementell zwischen der Faktentabelle und den Dimensionstabellen durch. Ein großes Problem dieser Methode ist, dass die Faktentabelle normalerweise viel größer als die Dimensionstabellen ist, was zu vielen großen Zwischenergebnissen führen kann. Dies kann die Abfrageleistung erheblich beeinträchtigen.

**Cross-Join Processing Plan**: Bei dieser Methode wird zunächst ein Kreuzprodukt aller Dimensionstabellen erstellt und dann mit der Faktentabelle gejoint. Da die Dimensionstabellen in der Regel viel kleiner sind, kann dies effizienter sein als die naive Strategie. Allerdings kann es auch hier nach einigen kartesischen Produkten zu großen Zwischenergebnissen kommen, die die Leistung beeinträchtigen.

**Semi-Join Processing Plan:** Der Semi-Join-Plan zielt darauf ab, die Tupel, die aus der Faktentabelle gelesen werden müssen, vor dem Join zu beschränken. Durch die Verwendung von Semi-Joins werden relevante RIDs (Row IDs) der Faktentabelle identifiziert, und nur jene Tupel aus der Faktentabelle werden ausgewählt, die in der resultierenden RID-Liste enthalten sind. Dieser Ansatz kann helfen, die Größe der Zwischenergebnisse zu reduzieren und die Leistung zu verbessern.

## Indices

Ein **Bitmap-Index** besteht aus mehreren Bit-Vektoren, wobei jeder Bit-Vektor für einen eindeutigen Wert einer Spalte steht. Wenn das Bit an Position n im Bit-Vektor auf "1" gesetzt ist, entspricht dies dem Wert, den der Bit-Vektor repräsentiert, in der Zeile n der Tabelle. Bitmap-Indizes können effizient multidimensionale Abfragen unterstützen, indem sie **Bit-Vektoren** verschiedener Attribute mit Bitoperationen (AND, OR, NOT) kombinieren.

Das Ziel von **decomposed Bitmap-Indizes** ist es, den Speicheraufwand zu reduzieren, indem Zahlen basierend auf ihren "Ziffern" zerlegt werden. Für jede Ziffer eines Zahlenwerts werden Bit-Vektoren erstellt. Dies reduziert die Anzahl der benötigten Bit-Vektoren erheblich, macht Abfragen jedoch teurer, da mehrere Bit-Vektoren kombiniert werden müssen, um eine Abfrage zu beantworten.

**Range encoded Bitmap-Indizes** zielen darauf ab, den Aufwand für Bereichsabfragen zu verringern. Anstatt für jeden einzelnen Wert in einem Bereich einen Bit-Vektor zu haben, wird ein Bit-Vektor für

"größer gleich" einem bestimmten Wert v verwendet. Dies reduziert die Anzahl der Bit-Vektoren, die für die Beantwortung von Bereichsabfragen benötigt werden, und ermöglicht eine effizientere Ausführung von Bereichsabfragen.

**Join-Indizes** sind spezielle Indizes, die dazu dienen, die Ausführung von Join-Operationen zwischen zwei oder mehr Tabellen zu beschleunigen. Sie speichern vorberechnete Verbindungen zwischen Tabellen, um die Zeit zu reduzieren, die für das Durchführen von Joins benötigt wird.

**Materialised Views** sind vorberechnete Ergebnisse von Abfragen, die physisch gespeichert werden. Sie dienen dazu, die Ausführungszeit von Abfragen zu verbessern, indem sie wiederholte Berechnungen vermeiden. Abfragen können direkt auf die vorberechneten Ergebnisse in der materialisierten Sicht zugreifen, anstatt die Daten jedes Mal neu zu berechnen.

## OLAP Query Execution

**Intra-Query Parallelismus**: Relationale Operatoren werden parallel auf verschiedenen Partitionen ausgeführt, um die Antwortzeit einer Abfrage auf große Datenmengen zu verringern.

**Parallel Sort**: Beinhaltet das lokale Sortieren von Partitionen, das Re-Partitionieren basierend auf dem Sortierschlüssel (Shuffling) und das erneute Sortieren der re-partitionierten Daten, eventuell mit Merge-Sort.

**Data Shuffling:** Implementiert als Send-Receive-Operator in verteilten DBMS, um Daten basierend auf dem Sortierschlüssel neu zu partitionieren. Es gibt verschiedene Shuffeling Strategien: Range, Hash-Partitioning und Replication, N:M und N:1 (Merge).

**Parallelaggregation:** Lokale Aggregation von Daten, gefolgt von einer N:1 Shuffle-Operation (Merge) und einer nachgelagerten Aggregation. Die Parallele Aggegation kann auch mit einem Group-By funktionieren, dann wird nach der lokalen Aggregation mittels dem Group-By Repartitioned.

**Symmetric Repartitioning Join:** Beide Tabellen A und B werden basierend auf dem Join-Schlüssel neu partitioniert, um eine gleichmäßige Verteilung für den Join zu gewährleisten.

**Asymmetric Repartitioning Join:** Nur eine der Tabellen (z.B. B) wird basierend auf dem Join-Schlüssel neu partitioniert, um sie der Partitionierungsstruktur der anderen Tabelle (A) anzupassen.

**Replication-/Broadcast-Based Join:** Alle Tupel der kleineren Tabelle (z.B. Tabelle A) werden an alle Knoten repliziert, um lokale Joins zu ermöglichen.

**Semi-Join Reduction**: Reduziert die zu versendenden Daten von Knoten 1 zu Knoten 2 auf die notwendigen Tupel, die einen Join-Partner auf der Gegenseite haben. Funktioniert mit Symmetric, Asymmetric und Replication Join. Diese Strategie funktioniert gut, wenn der Semi-Join die Größe der zu verbindenden Tabellen reduziert. Semi-Join kostspieliger als Versand der gesamten Partition R, falls die meisten Zeilen aus R sowieso für den Join mit S benötigt werden.

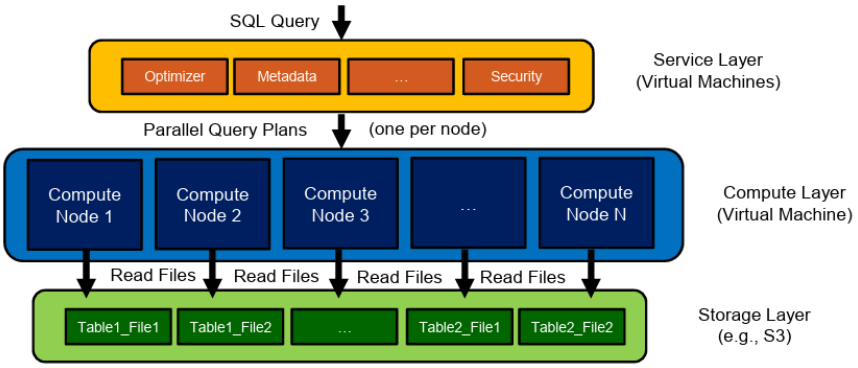
**Hypercube Shuffling**: Organisiert Worker als Hypercube, wobei jede Dimension einem Join-Schlüssel entspricht, und ermöglicht das Shuffling derselben Tupel zu mehreren Knoten, um lokale Joins effizient durchführen zu können.

Die optimalen Anteile ist die Kombination mit den geringsten Netzwerkkosten (Also geringste Anzahl der zu shuffelnden Tupel)

## OLAP-Architektur

- Storage Layer: Die Datenbank ist in Partitionen (Dateien) geteilt. Diese Dateien sind in einem skalierbaren Storage Service gespeichert (PaaS).

- Compute Layer: Eine Menge an virtuellen Maschinen (IaaS), die Rechennodes lesen die Dateien aus dem Storage Layer um sie zu verarbeiten Der Query und der Service Layer verwenden zwei verschiedene Sätze an Maschinen.
    - Query Layer: Verantwortlich Query Pläne und Transaktionen parallel auszuführen
    - Service Layer: Compiler und Optimierer, Metadaten, Sicherheit etc




## OLTP (Online Transaction Processing)

OLTP-Systeme sind darauf ausgelegt, eine große Anzahl an kurzen Onlinetransaktionen auszuführen. OLTP wird häufig für eine große Anzahl an Nutzern verwendet, welche einfache Queries ausführen. OLTP verwendet Inter-Query Parallelismus.

### Partitionsschema

Die Partitionierungsstrategien für OLTP ähneln denen von OLAP, mit dem Ziel, Netzwerkkommunikation zu vermeiden, beispielsweise durch Co-Partioning von Daten. Dies ist wichtig, um die Latenz bei Transaktionen zu minimieren, die Daten auf mehreren Servern aktualisieren müssen. Für komplexe OLTP Workloads kann man zum Beispiel Schism verwenden:

Schism ist eine Partitionierungsmethode für OLTP-Systeme, die darauf abzielt, die Anzahl verteilter Transaktionen zu minimieren und die Leistung des Systems zu optimieren. Der Prozess von Schism umfasst mehrere Schritte:

1. Aufbau eines Graphen aus einem Workload Trace:
    a. Knoten: Die Knoten im Graphen repräsentieren Tupel, die in dem Trace erfasst wurden
    b. Kanten: Kanten verbinden Tupel, auf die von derselben Transaktion zugegriffen wurde. Das bedeutet, wenn zwei Tupel A und B von derselben Transaktion verwendet werden, wird eine Kante zwischen A und B im Graphen eingefügt.
2. Partitionierung sollte verteile Transaktionen minimieren: Die Idee ist, dass durch die Anwendung eines Min-Cut-Algorithmus auf den Graphen die Anzahl der verteilen Transaktionen minimiert wird. "Min-Cut" bedeutet hier, den Graphen in n disjunkte Teilmengen zu schneiden und dabei die Summe der Gewichte der geschnittenen Kanten zu minimieren. Die Gewichte auf den Kanten entsprechen der Anzahl der Transaktionen, die die verbundenen Tupel gemeinsam nutzen.

## Distributed OLTP

Das Prinzip einer Transaktion ist eine Sequenz von Sukzessiven DB Operationen die von einem BOT … EOT umfasst sind (Commit / Abort). Das DBMS garantiert dabei ACID: Atomicity, Consistency, Isolation und Durability. Verteilte Transaktionen sind der Access oder das Update von Daten auf mehr als einem Node. Wichtig ist, dass hier auch die ACID Bedingungen eingehalten werden.

- Distributed Commit, um Atomicity sicherzustellen
- Distributed Concurrency, um Isolation sicherzustellen
- Replication Protocol, um Sicherzustellen, dass Daten zuverlässig auf den Replikas gespeichert werden (Durability)

Herausforderungen:

- Ergebnisse müssen konsistent mit der Sicht des Nutzers ein
- Transaktionen müssen dieselbe Abfolge an unterschiedlichen Standpunkten haben
- Die Nachrichten müssen synkronisiert sein aber jeder Computer hat seine eigene lokale Zeit und Nachrichten können mit unvorhersehbaren Delays ankommen oder gar nicht ankommen
- Die Lösung hierbei ist eine logische Zeit festzulegen, also werden die Events immer in Abhängigkeit vom vorherigen Event gemessen (Lamport Clocks)
    - Aber Nodes können ausfallen, Nachrichten können verloren gehen und Performanz ist abhängig von der langsamsten Nachricht

Atomic Commitment

- Commit Protokoll garantiert Atomicity in verteilten DBMS
- Das Protokoll „versteht“ die Semantik eines TX Abort/Commit

Consensus Protokolle

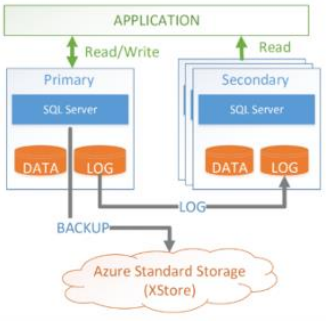
- Das Konsensproblem ist das Problem, eine Gruppe von Knoten in einem verteilten System dazu zu bringen, sich auf etwas zu einigen (z.B. ein Ergebnis eines tx = Commit oder Abbruch)
- Konsensprotokolle können erweitert werden, um als atomares Commit-Protokoll verwendet zu werden (Logik wird hinzugefügt, um zu entscheiden, ob Commit oder nicht)

## OLTP-Architekturen

Für OLTP gibt es verschiedene Architekturen, welche sich durchgesetzt haben:

### Non-partitioned, replicated

Baut auf ein Traditionelles Single-Node Design



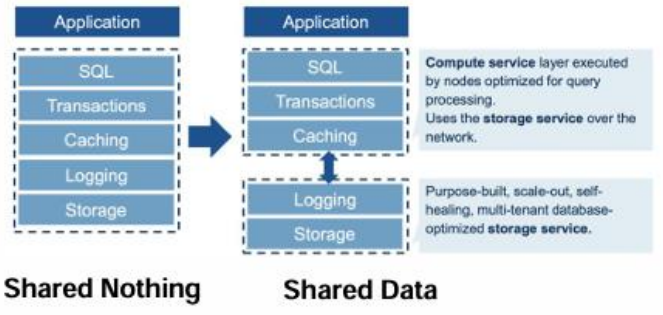

- Datenbank nicht partitioniert
- Primärer Node verarbeitet alle Update Transaktionen auf der gesamten Datenbank
- Verteilt den Update-Log auf alle folgenden Replikationen
- Primäre Replikation macht regelmäßige Back-Ups für Katastrophen Resilienz
- Folgende Replikationen verarbeiten nur read-only Queries für load-balencing

Vorteile:

- Nur kleine Änderungen am DBMS notwendig
    - Services sind Stabil und erwachsen
    - Weit verbreitet Architektur
- Hohe Performanz und Verfügbarkeit
    - Jeder Rechennode hat eine volle, lokale Kopie der Datenbank
    - Ein Failover von einem Primary zu einem Secondary Node ist problemlos möglich

Nachteile:

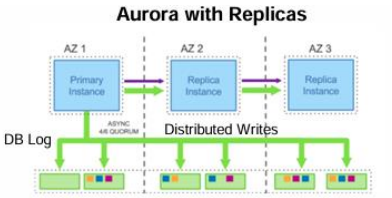
- Die Datenbankgröße kann nicht größer als die Datenspeichergröße eines einzelnen Nodes werden
- O(size-of-data) Operationen machen Probleme, diese müssen auf einem einzelnen Server ausgeführt werden
- Scale-Up und Scale-Down sind problematisch. Ein neuer Node braucht eine komplette Kopie der Datenbank

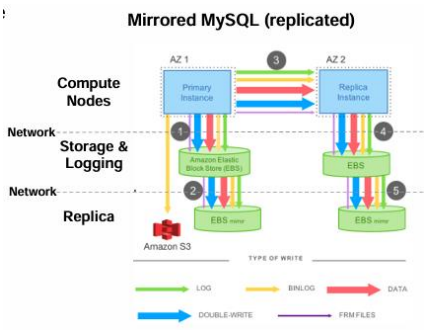



## Non-partitioned, shared Data

Ähnlich wie non-partitioned aber Daten von Compute entkoppelt und als seperate skalierbare Services.

Naive Lösung: Einfach DBMS Architektur verwenden und Speicher von den oberen Layern trennen. Führt aber zu einer hohen Anzahl an Writes von jedem Node, viele redundate Writes und Logs müssen per Netzwerk verschickt werden, ist also langsamer als non-partioned, replicated. Die Lösung ist, dass die Computenodes nur den Log in den Storage Layer schreiben.
Der Storage Layer rekonstruiert die Pages dann aus dem Log (im Hintergrund), sorgt für deutlich weniger IOs zwischen Compute und Storage layer




Vorteile:

- Einfach Hoch- und Runterskalierbar mit der Last
- Einfach Hoch- und Runterskalierbar mit der Größe der Datenbank, unerreichbare Nodes könne einfach ausgetauscht werden
- Einfacher Failover des primären Nodes auf einen größeren Node

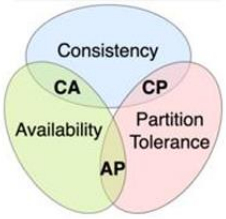
## Replikation für Fehlertoleranz

Replikas können, wie State Machines betrachtet werden, Updates sind Transitionen zwischen States und die Operationen sind deterministisch. Ein Update muss entweder auf allen Replikas angewandt werden oder auf keinem. Ein Node hat dabei die Primärrolle und bekommt die Updaterequests vom Nutzer und kommuniziert mit den Replikas, um dort alle Veränderungen anzuwenden.
Folgendermaßen läuft eine synchrone Replikation ab:

1. Primary bekommt Updaterquest vom Nutzer
2. Primary fordert das Update (Log)
3. Primary spiegelt den Log auf die Backups
4. Backups führen den Log aus und senden ein ACK zum Primary
5. Primary führt den Log aus

**Rekonfiguration:** Wenn ein Backup fehlschlägt, muss eventuell ein neuer Node hinzugefügt werden, hierfür braucht es ein Protokoll, damit der neue Node die Log Operationen hat bevor es weitergeht.

**Leader Election:** Wenn der Primary fehlschlägt, muss ein neuer ausgewählt werden. Hierfür wird ein Konsens Protokoll verwendet damit sich alle Backups auf eine neue Primary einigen können




**CAP-Theorem:** In einem verteilten System können nur zwei von den 3 Eigenschaften rechts gleichzeitig erreicht werden

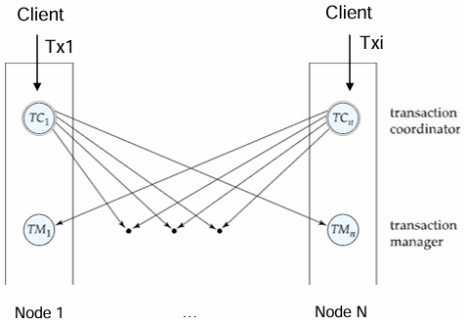
**Reliable Totally Ordered Multicast:** Verwendet, um alle Operationen an alle Nodes zu verteilen

**Distributed Consensus Protocols:** Diese bieten eine Möglichkeit für alle Nodes um sich auf eine Fehlertoleranz zu einigen. Konsens wird häufig für die Replikation verwendet, um eine zuverlässige geordnete Zustellung zu allen Teilnehmern sicherzustellen und hat einen Mechanismus um viele verschiedene Fehler zu behandeln.

**Eventual Consistency**: Eventuell, alle Reads werden konsistent, in der Zwischenzeit sollen die daten möglichst lange gestallt werden.

**Strong Consistency**: Alle Reads sind immer up to date, Immer Primary lesen, Primary ist das Bottleneck, oder großteil der Replikas lesen und neusten Wert lesen.

## Distributed OLTP Processing Model




Jede Node hat einen Transitions Koordinator (TC): Clients reicht Transaktion beim TC ein, TC koordiniert die gesamte Ausführung in dem er die Kommandos an alle TX Manager sendet

Jeder hat einen lokalen Transaktions Manager (TM), Tm führt alle Read/Write Operationen auf den lokalen Daten aus und entscheidet lokal ob Commit oder Abort.

## Commit Protokolle

**Atomic commit protocol:** Diese Protokolle werden verwendet, um Atomicity über alle Standorte sicherzustellen. Eine Transaktion, welche auf mehreren Standorten ausgeführt wird, muss entweder zu allen Standorten committed werden oder auf allen abgebrochen werden.

Das **Two Phase commit (2PC)** protocol wird häufig als atomic commit protocol in verteilten DBMS eingesetzt. Die Ausführung des 2PC-Protokolls wird vom Transaktionskoordinator eingeleitet, nachdem die letzte Operation der Transaktion ausgeführt wurde. Das Protokoll bezieht alle lokalen Standorte ein, an denen Teile der Transaktion ausgeführt werden. 2PC beginnt, sobald die Transaktion zu Ende ausgeführt wurde (d.h. alle Operationen abgeschlossen sind) und eine Commit-Entscheidung getroffen werden muss. Das Protokoll hat zwei Phasen

1. Vorbereitungsphase: Frage an alle Standorte, ob sie die Transaktion durchführen können (oder abbrechen müssen).
2. Commit-or-Abort Phase: Senden der Entscheidung über Zusage oder Abbruch an alle Knotenpunkte

Optimierungen des 2PCs:

**Early Prepare Voting of nodes:** Wenn Sie einen Vorgang an einen entfernten Knoten senden, von dem Sie wissen, dass es der letzte ist, den Sie dort ausführen, dann wird dieser Knoten auch sein Votum (Commit oder Abort) für die Vorbereitungsphase mit dem Abfrageergebnis zurückgeben.

**Early Acknowledgement to client:** Wenn alle Knoten für die Übermittlung einer Nachricht korrekt sind, kann der Koordinator dem Kunden eine Bestätigung senden, dass seine Nachricht erfolgreich war, bevor die Übertragungsphase abgeschlossen ist.

## Cloud OLTP

Traditionelle Datenbanken sind zwar schnell aber haben keine Fehlertoleranz und lassen sich schwer skalieren.

**EC2**, kurz für Amazon Elastic Compute Cloud, ist ein zentraler Bestandteil von Amazons Cloud-Computing-Plattform, Amazon Web Services (AWS). EC2 bietet skalierbare Rechenkapazitäten in der AWS-Cloud. Nutzer können virtuelle Maschinen, sogenannte Instances, mit verschiedenen Konfigurationen von CPU, Speicher, Speicherplatz und Netzwerkkapazitäten starten, um jede beliebige Serveranwendung in der AWS-Cloud auszuführen. EC2 bietet Flexibilität und eine Vielzahl von Optionen, die es Nutzern ermöglichen, die Rechenressourcen an ihre Anforderungen anzupassen.

**EBS**, oder Amazon Elastic Block Store, ist ein Block-Speicherdienst, der für die Verwendung mit Amazon EC2 entwickelt wurde, um persistente Speicherlösungen zu bieten. EBS-Volumes können an EC2-Instances angehängt werden und funktionieren wie herkömmliche Festplatten, die an einen physischen Computer angeschlossen sind. EBS ermöglicht es Nutzern, Daten dauerhaft zu speichern, auch über die Lebensdauer der EC2-Instances hinaus, und unterstützt eine Vielzahl von Arbeitslasten, darunter Datenbanken, Dateisysteme und Anwendungen, die eine hohe Verfügbarkeit und Zuverlässigkeit erfordern.

**RDS**, der Amazon Relational Database Service, ist ein verwalteter Datenbankdienst, der das Einrichten, Betreiben und Skalieren einer relationalen Datenbank in der Cloud vereinfacht. RDS unterstützt mehrere Datenbank-Engines. Mit RDS können Nutzer sich auf die Anwendungsentwicklung konzentrieren, ohne sich um Aufgaben wie Datenbankverwaltung, Patching, Backups und Skalierung kümmern zu müssen. RDS bietet automatisierte Backups, Patch-Management und Failover-Funktionen, um die Datenbankverfügbarkeit und -zuverlässigkeit zu gewährleisten.

**Amazon Aurora** ist eine relationale Datenbank, die für die Cloud optimiert ist und Teil des Amazon RDS ist. Die Ziele von Aurora umfassen:

- Hochleistung und Skalierbarkeit: Aurora bietet eine höhere Leistung als standardmäßige MySQL- und PostgreSQL-Datenbanken durch eine optimierte, cloud-native Architektur.
- Verfügbarkeit und Haltbarkeit: Aurora ist darauf ausgelegt, hochverfügbar und fehlertolerant zu sein, mit automatischem Failover, Backup und Wiederherstellung.
- Kosteneffizienz: Durch die automatische Skalierung der Speicherkapazität und die On-Demand-Provisionierung der Rechenressourcen bietet Aurora eine kosteneffiziente Lösung.

**Log Shipping** ist eine Technik zur Datenreplikation und Disaster Recovery, bei der Transaktionslog-Einträge kontinuierlich von einer primären Datenbank zu einer sekundären Datenbank kopiert und dort angewendet werden. Dies ermöglicht eine nahezu Echtzeit-Datenreplikation und kann zur Failover-Unterstützung verwendet werden.

**Quorum** bezieht sich auf die Mindestanzahl von Mitgliedern (z.B. Knoten in einem Cluster, Repliken in einem Replikationsschema), die erreicht werden muss, um eine konsistente Operation oder Entscheidung in einem verteilten System durchzuführen. In einem Datenbankkontext kann Quorum dazu beitragen, die Konsistenz von Daten über Repliken hinweg zu gewährleisten und Split-Brain-Szenarien zu vermeiden, in denen zwei Teile eines Clusters unabhängig voneinander und inkonsistent arbeiten.

Log Shipping und Quorum arbeiten zusammen, um asynchrone Transaktionsverarbeitung zu ermöglichen. Um auf der Speicherseite zu skalieren, verwendet man Schutzgruppen auf den Daten. Schnelle Recovery ist gerade in großen Systemen wichtig, da Fehler auch hier skalieren, gerade in der Cloud dauert es sehr lange die Daten wiederherzustellen Aufgrund der limitierten Netzwerkgeschwindigkeit. Desegregation (Trennen von Speicher + Ausführung, siehe shared Data) ist der Schlüssel aber es muss mit smartem Computing, Log Servern und Page Servern geschehen.

### Multi-Writer OLTP Databases in the Cloud

Google Spanner ist eine hochverfügbare, globale Datenbankdienstleistung, die die Vorteile relationaler Datenbankstrukturen mit der Skalierbarkeit von NoSQL-Systemen verbindet. Sie bietet eine starke Konsistenz über Rechenzentren hinweg und unterstützt SQL-Abfragen. Die Architektur von Google Spanner kombiniert Elemente relationaler Datenbanken mit der Skalierbarkeit von NoSQL-Systemen. Ein Schlüsselkonzept in Spanner ist der "Replicated Single Shard", was bedeutet, dass jede Datenpartition (Shard) über mehrere Standorte repliziert wird, um hohe Verfügbarkeit und Haltbarkeit zu gewährleisten. Spanner verwendet Synchronisationsmechanismen, um eine starke Konsistenz über alle Replikate hinweg zu gewährleisten.

**2PC** ist ein Protokoll zur Erreichung der atomaren Commitment-Eigenschaft in verteilten Systemen. Google Spanner verwendet 2PC, um sicherzustellen, dass eine Transaktion entweder vollständig über alle beteiligten Shards und deren Replikate hinweg ausgeführt wird oder vollständig abgebrochen wird, wenn ein Fehler auftritt. Dies gewährleistet die Datenintegrität und Konsistenz über die verteilte Datenbank hinweg.

**2PL** ist eine Sperrtechnik, die verwendet wird, um die Serialisierbarkeit von Transaktionen in Datenbanksystemen zu gewährleisten. Während der ersten Phase, der Sperrphase, werden alle benötigten Sperren von der Transaktion erworben. In der zweiten Phase, der Freigabephase, werden alle Sperren freigegeben. Obwohl Google Spanner 2PC für das Commitment von Transaktionen verwendet, ist nicht explizit dokumentiert, dass es 2PL für die Sperrverwaltung nutzt. Stattdessen setzt Spanner auf eine Kombination aus Sperrfreiheit und dem Einsatz von Zeitstempeln, um die Serialisierbarkeit und externe Konsistenz zu gewährleisten.

**Paxos** ist ein Konsensprotokoll, das in verteilten Systemen verwendet wird, um einen Konsens unter einer Gruppe von Replikaten zu erreichen, auch wenn einige Replikate ausfallen oder unzuverlässig sind. Google Spanner nutzt eine Variante von Paxos, um Replikate innerhalb eines Shards zu synchronisieren und sicherzustellen, dass alle Replikate einen konsistenten Zustand aufweisen. Paxos ermöglicht es Spanner, hohe Verfügbarkeit und Datenkonsistenz zu bieten, selbst in einem Umfeld, in dem Netzwerkpartitionen oder Serverausfälle auftreten können.

## Query Optimierung

Query Optimierung kann Kosten oder Regelbasiert geschehen.

**Regelbasierte Optimierung:** Bei der regelbasierten Optimierung wird der logische Plan einer Abfrage mithilfe eines Satzes von Umschreibungsregeln transformiert. Diese Regeln werden angewendet, um die Struktur des Abfrageplans ohne Berücksichtigung der spezifischen Ausführungskosten zu verbessern. Ein Beispiel für eine solche Regel ist das Herunterdrücken von Selektionen (Selection Pushdown), bei dem Selektionsoperationen so weit wie möglich nach unten im Abfrageplan verschoben werden, um die Menge der zu verarbeitenden Daten so früh wie möglich zu reduzieren.

**Kostenbasierte Optimierung:** Nachdem die regelbasierte Optimierung angewendet wurde, kommt die kostenbasierte Optimierung zum Einsatz. Dabei wird versucht, den optimalen Ausführungsplan basierend auf geschätzten Ausführungskosten zu finden. Zwei wichtige Schritte sind hierbei die

Bestimmung der optimalen Join-Reihenfolge im lokalen Plan und die Auswahl der physischen Operatoren für die Ausführung der verschiedenen Teile des Plans. Die Kosten werden typischerweise anhand von Metriken wie der erwarteten Anzahl von Disk-Zugriffen oder der Größe der Zwischenergebnisse geschätzt.

- **Physische Kosten Schätzung**: Die physische Kosten Schätzung bezieht sich auf die Auswahl eines physischen Operators für jeden logischen Operator nach der Join-Reihenfolge mit dynamischer Programmierung. Zu berücksichtigende Faktoren sind sequentielle vs. zufällige I/O-Kosten und verfügbare Indizes. Die Auswahl des physischen Operators führt zu einem physischen Plan, der auf den Kardinalitätsschätzungen basiert.
- **Join-Ordering**: Die Join-Reihenfolge ist entscheidend, da dieselbe Abfrage mit verschiedenen Join-Reihenfolgen zu sehr unterschiedlichen Laufzeiten führen kann. Die Kosten eines Plans werden als Funktion modelliert, die es erlaubt, die Laufzeitkomplexität eines Abfrageplans zu schätzen. Ein einfaches Kostenmodell für logische Planalternativen ist die Summe der Zwischenergebnisgrößen (Kardinalitäten).
    - **Search Space**: Der Suchraum wächst sehr schnell mit der Anazhl der tabellen, die in einer Query gejoint werden. Die Anzahl der möglichen Join Bäume für n+1 Leaves (also Tabellen) ist durch die Catalan Nummer gegeben: $C_{n-1} = (2n)!/((n+1)! \times n!)$, Jeder Baum hat $(n+1)!$ Tabellen Permutationen was zu $\frac{(2n)!}{n!}$möglichen Join Tress führt.
    - **Dynamic Programming:** Methode, die darauf abzielt, eine Näherungslösung für ein Optimierungsproblem zu finden, indem sie das Problem in kleinere Teilprobleme zerlegt und die Lösungen der Teilprobleme verwendet, um die Lösung des Gesamtproblems zu konstruieren. Im Kontext der Join-Reihenfolge bedeutet dies, dass Pläne mit n Tabellen aus optimalen Plänen mit n−1 Tabellen zusammengesetzt werden.

## Cardinality Estimation & DeepDB

Traditionelle Techniken zur Kardinalitätsschätzung machen mehrere vereinfachende Annahmen, die zu großen Fehlern führen können. Dazu gehören die Annahme einer gleichmäßigen Datenverteilung (wenn keine Histogramme verwendet werden) und die Annahme, dass alle Spalten unabhängig (unkorreliert) sind.

Die **Selektivität** sel(x) beschreibt das Verhältnis von Output zu Input eines Operators Die geschätzte Selektivität wird verwendet um die zwischenzeitliche Kardinalität zu schätzen.

| Formel | Beschreibung |
|---|---|
| $sel(p) \coloneqq \frac{\lvert\sigma_p(R)\rvert}{\lvert R\rvert}$ | Selektivität für Selection Operator |
| $\lvert\sigma_p(R)\rvert = sel(p) \cdot \lvert R\rvert$ | Geschätzte Output Kardinalität |
| $sel(R \bowtie S) \coloneqq \frac{\lvert R \bowtie S\rvert}{\lvert R \times S\rvert} = \frac{\lvert R \bowtie S\rvert}{\lvert R\rvert \cdot \lvert S\rvert}$ | Selektivität für Join Operator |
| $\lvert R \bowtie S\rvert = sel(R \bowtie S) \cdot \lvert R\rvert \cdot \lvert S\rvert$ | Geschätzte Output Kardinalität |
| $sel(att = constant) = \frac{1}{\lvert att\rvert}$ | Selektivität für Predikat von $\sigma_p$, $\lvert att\rvert$ verschiedene Werte für das Attribut |
| $sel(att\ in\ range) = \frac{\lvert range\rvert}{\lvert att\rvert}$ | Selektivität für Predikat von $\sigma_p$, $\lvert range\rvert$ verschiedene Werte für die Range |
| $sel(p_1\ AND\ p_2) = sel(p_1) \cdot sel(p_2)$ | Selektivität für komplexe Prädikate |
| $sel(p_1\ OR\ p_2) = sel(p_1) + sel(p_2) - sel(p_1) \cdot sel(p_2)$ | Selektivität für komplexe Prädikate |

| | |
|---|---|
| $sel(R \bowtie_{R.a=S.b} S) \coloneqq \frac{\lvert S\rvert}{\lvert S\rvert \cdot \lvert R\rvert}$ | Selektivität für Equi-Join, bei dem R und S Basetable sind und R.a der primary Key und S.b der foreign Key ist. |
| $\lvert R \bowtie_{R.a=S.b} S\rvert \coloneqq \lvert S\rvert \cdot \lvert R\rvert \cdot \mathrm{sel}(\mathrm{R} \bowtie \mathrm{S}) = \lvert \mathrm{S}\rvert$ | Geschätzte Output Kardinalität |
| $sel(R \bowtie_{R.a=S.b} S) \coloneqq \frac{1}{\max(\lvert a\rvert,\lvert b\rvert)}$ | Selektivität für Equi-Join, bei dem R und S keine Basetable sind (z.B. nach Filter) |
| $\lvert R \bowtie_{a=b} S\rvert = \lvert R\rvert \cdot \lvert S\rvert \cdot \frac{1}{\max(\lvert a\rvert,\lvert b\rvert)}$ | Geschätzte Output Kardinalität |
| $\lvert R_1 \bowtie R_2 \bowtie R_3\rvert = \lvert (R_1 \bowtie R_2) \bowtie R_3\rvert$ | Mehrere Joins |

Beispiel:

Selektivität für $\sigma_{Height=tall}(Person)$ berechnen, wir haben $|Height| = 2$ und $|Person| = 100.000$, somit gilt $sel(Height = tall) = \frac{1}{|Height|} = \frac{1}{2} = 0.5$. Die erwartete Output-Größe ist somit $\left|\sigma_{Height=tall}(Person)\right| = sel(Height = tall) \cdot |Person| = \frac{1}{2} \cdot 100.000 = 50.000$.

**Learned Cardinality:** Die Idee hinter der gelernten Kardinalitätsschätzung ist die Verwendung von maschinellen Lernmodellen, um Datenverteilungen detaillierter zu erfassen und Korrelationen zwischen Spalten besser abzubilden. Ein Ansatz besteht darin, neuronale Netzwerke zu verwenden, um Kardinalitäten vorherzusagen.

**Histogramme:** Werden verwendet, um die Verteilung der Datenwerte in einer Spalte zu modellieren. Ein Histogramm kann dem Optimierer helfen zu verstehen, wie die Daten verteilt sind und wie viele Zeilen voraussichtlich eine bestimmte Abfragebedingung erfüllen.

**Sampling:** Beim Sampling wird eine kleine Stichprobe der Daten analysiert wird, um Rückschlüsse auf die gesamte Datenmenge zu ziehen. Dies kann eine effiziente Möglichkeit sein, nützliche Schätzungen zu erhalten, ohne alle Daten scannen zu müssen.

**Parameterized Distributions:** Wahrscheinlichkeitsverteilungen (z.B. Gauß Verteilung mit Mittelwert und Standardabweichung) verwenden, um die Daten zu modellieren

**DeepDB:** DeepDB ist ein Ansatz, der auf Daten und nicht auf Abfragen lernt. Es verwendet Relationale Sum-Produkt-Netzwerke (RSPNs), die speziell für relationale DBMS entwickelt wurden. RSPNs lernen Datenverteilungen innerhalb und zwischen Tabellen und ermöglichen die Unterstützung beliebiger Joins durch die Kombination von RSPNs. Ein Ensemble von RSPNs wird für jede Datenbank verwendet.

### RSPN

**Relational-Sum-Product-Networks:** RSPNs sind Modelle, die für relationale DBMS maßgeschneidert sind. Sie lernen die Datenverteilungen innerhalb und zwischen den Tabellen und unterstützen beliebige Joins, indem sie RSPNs kombinieren. Ein Ensemble von RSPNs wird pro Datenbank eingesetzt, um eine umfassende und genaue Kardinalitätsschätzung zu ermöglichen.

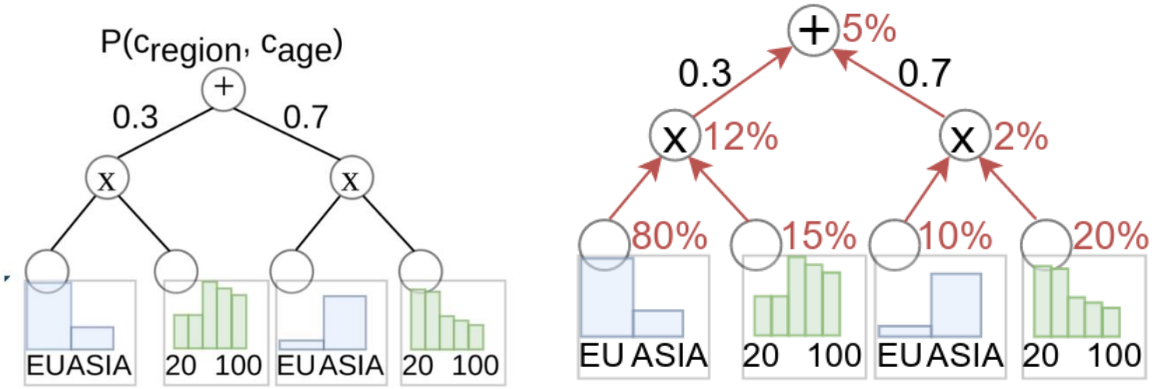
RSPNs kombinieren Summen- und Produktknoten in einem Baumgraphen, um komplexe Verteilungen zu modellieren.

- **Produktknoten (X):** Diese Knoten repräsentieren die Faktorisierung von Wahrscheinlichkeiten, was bedeutet, dass die Wahrscheinlichkeiten von Ereignissen, die sie repräsentieren, als unabhängig angenommen und daher multipliziert werden können.
- **Summenknoten (+):** Diese Knoten repräsentieren Mischungen von Verteilungen. Die Kanten von einem Summenknoten zu seinen Kindern haben Gewichte, die die Wahrscheinlichkeit repräsentieren, dass das Kind ausgewählt wird.

- **Blätter des Baumes:** Die Blätter eines RSPN sind Verteilungen über einzelnen Variablen oder Gruppen von Variablen. Sie können einfache Verteilungen wie Gaußsche Verteilungen für kontinuierliche Variablen oder Multinomialverteilungen für kategoriale Variablen sein.

Beispiel zu RSPNs:

EU, unter 30, die genauen Wahrscheinlichkeiten der Blätter müssen angegeben sein, dann kann man den Baum leicht berechnen




## Distributed Optimierung

Bei Distributed Query Optimization geht es darum, den effizientesten Weg zur Ausführung einer Abfrage in einem verteilten Datenbankmanagementsystem (DBMS) zu finden, wobei die Daten über mehrere Knoten verteilt sind. Das Hauptproblem dabei ist, dass die Anzahl der möglichen parallelen Ausführungspläne, aus denen man wählen kann, viel größer ist als die Anzahl der Pläne in einem Single-Node-System. Daher wird üblicherweise folgendes Verfahren für parallele DBMS verwendet:

1. Wähle zuerst den effizientesten Single-Node-Plan: Hierbei wird der traditionelle Optimierer genutzt, um den besten Plan für die Ausführung auf einem einzelnen Knoten zu bestimmen.
2. Parallelisiere jede Operation über partitionierte Daten: Anschließend wird für jede Operation im Plan ein paralleler Operator gewählt (z.B. ein partitionierter Join), der die Netzwerkkosten minimiert. Dies bedeutet, dass die Daten so über die Knoten verteilt werden, dass die notwendigen Datenübertragungen zwischen den Knoten möglichst geringgehalten werden, um die Abfrage effizient ausführen zu können.

| Join Algorithmus | Total Network Cost (in # of tuples) |
|---|---|
| Symmetric Repartition | $\frac{\lvert R\rvert\times(n-1)}{n} + \frac{\lvert S\rvert\times(n-1)}{n}$ |
| Asymmetric Repartition | $\frac{\lvert R\rvert\times(n-1)}{n}$ |
| Replicate (if R iss maller Table) | $\lvert R\rvert \times (n-1)$ |

# Query Plan

**Query Compilation:** Bei der Abfragekompilierung wird eine SQL-Abfrage in einen kanonischen logischen Plan umgewandelt. Dieser Prozess übersetzt die hochsprachliche SQL-Anfrage in eine interne Repräsentation, die für das Datenbanksystem verständlich ist. Der logische Plan enthält die grundlegenden Operationen wie Selektion, Projektion und Joins, die erforderlich sind, um die Abfrage zu erfüllen, ohne dabei spezifische Details zur Ausführung zu berücksichtigen.

Die SQL Notation kann hier nachgeschlagen werden:
http://infolab.stanford.edu/~ullman/fcdb/aut07/slides/ra.pdf

**Query Optimization:** Die Abfrageoptimierung ist der Prozess, bei dem der kanonische logische Plan analysiert und in den besten "physischen Plan" für die Ausführung transformiert wird. Ziel ist es, den Plan zu finden, der voraussichtlich die minimale Laufzeit hat. Der Optimierer bewertet verschiedene Strategien zur Ausführung der Abfrage, wie beispielsweise die Reihenfolge von Joins, die Auswahl von Indizes und die Anwendung verschiedener Algorithmen für die Datenverarbeitung, um den effizientesten Ausführungsplan zu bestimmen. Optimierung kann Regelbasiert oder kostenbasiert stattfinden.

## Execution Models

**Materialized Execution:** Bei der materialisierten Ausführung berechnet jeder Operator seine vollständige Ausgabe, bevor der nächste Operator beginnt. Dies bedeutet, dass die Ergebnisse eines Operators komplett generiert und möglicherweise temporär gespeichert werden, bevor der nächste Schritt der Abfragebearbeitung fortgesetzt wird. Dieser Ansatz kann effektiv sein, wenn die Zwischenergebnisse mehrfach verwendet werden müssen, kann jedoch bei großen Datenmengen zu einem Overhead führen, da die Daten zwischen den Operationen gespeichert werden müssen.

**Pipelined Execution:** Bei der gepipelten Ausführung werden Tupel sofort an den nächsten Operator weitergegeben. Dies ermöglicht eine schnellere Verarbeitung, da nicht gewartet werden muss, bis ein Operator seine gesamte Ausgabe generiert hat, bevor mit der Verarbeitung der nächsten Stufe begonnen wird. Die gepipelte Ausführung ist besonders effektiv in Situationen, in denen die Daten schrittweise verarbeitet werden können und das Endergebnis schrittweise generiert wird, ohne dass alle Zwischenergebnisse vollständig materialisiert werden müssen.

## Physische Operatoren

**Table Scan:** Durchsucht die gesamte Tabelle und prüft jedes Tupel gegen eine Auswahlbedingung.

**Index Scan mit B+-Baum**: Durchläuft den Index und liest Seiten aus der Tabelle.

**Multi-Attribut Selektion**: Kombiniert mehrere einfache Prädikate mit AND/ODER.

**Nested-Loop Join**: Durchläuft jede Kombination von Tupeln in den beiden Relationen und prüft, ob sie zusammenpassen.

**Block Nested-Loop Join**: Verwendet blockbasierte Verarbeitung, um die Anzahl der Seitenzugriffe zu reduzieren.

**Index Nested-Loop Join**: Verwendet einen Index für die innere Tabelle, um nur relevante Zeilen für den Join zu holen.

**Sort-Merge Join**: Setzt voraus, dass beide Eingaben auf dem Join-Schlüssel sortiert sind.

**Hash Join**: Erstellt eine Hashtabelle auf der kleineren Eingabetabelle und prüft die Einträge der größeren Tabelle gegen diese Hashtabelle.

**Grace Hash Join**: Wird verwendet, wenn die Hashtabelle nicht in den Speicher passt. Beide Tabellen werden partitioniert und die Joins werden partitioniert durchgeführt.

**Simple Sort**: Alle Eingabedaten werden in temporären Speicher geladen und dort sortiert.

**Two-Way Merge Sort**: Jede Page wird individuell sortiert, danach werden 2 sortierte Pages zusammengeführt.

### Kosten

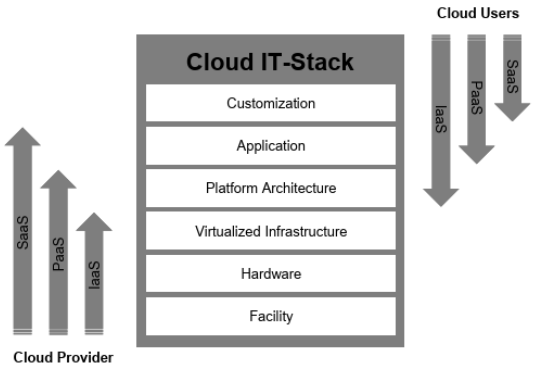
Die Ausführungskosten schätzen die Laufzeit eines Operators ab. Die Kosten werden oft als Anzahl der Seiten modelliert, die von der Festplatte gelesen oder darauf geschrieben werden müssen: Kosten = #SeitenLesen + #SeitenSchreiben. Darüber hinaus wird üblicherweise zwischen den Kosten für zufälligen und sequenziellen Zugriff auf Seiten unterschieden: Kosten = crand * #SeitenRand + cseq * #SeitenSeq (Die Konstanten crand und cseq müssen vom Datenbankadministrator eingestellt werden). Für die Einfachheit in der Vorlesung: Wir betrachten nur die Gesamtanzahl der Seiten. Annahmen und Notationen: Tabelle R hat M Seiten und m Tupel. Die Anzahl der Seiten M ist viel kleiner als die Anzahl der Tupel m.

| Physicher Operator | Kosten |
|---|---|
| Table Scan | $M$ (not sorted), $\log_2 M$ (sorted) |
| Index Scan mit B+-Baum | $\lceil \log_{index-fanout} m \rceil + 1$ |
| Multi-Attribut Selektion | Abhängig von den Prädikaten |
| Nested-Loop Join | $m + m \times n$ |
| Block Nested-Loop Join | $M + M \times N$ |
| Index Nested-Loop Join | $M + m \times \lceil \log_{fanout} n \rceil + 1$ |
| Sort-Merge Join | $M + N$ |
| Hash Join | $M + N$ |
| Grace Hash Join | Ähnlich zu HashJoin aber extra Partitonierung |
| Simple Sort | |
| Two-Way-Sort | $(\log_2 M + 1) \times 2 \times M$ |

# Cloud

Die Ziele der Cloud sind Skalierbarkeit, Elasitzität, Fehlertoleranz und das Pay-as-you-go Modell. Die Cloud gibt die Illusion von unendlicher Skalierbarkeit und das mit anpassen der Ressourcen, wenn Notwendig (nie zu wenig Ressourcen oder unnötig viele, kann sich Lastspitzen anpassen). Durch hohe Redundanz wird hohe Erreichbarkeit garantiert, Services heilen sich selbst.

**Definition:** Cloud computing is a model for enabling ubiquitous, convenient, on demand network access to a shared pool of configurable computing resources (e.g., … servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction.




Die Cloud bietet SaaS (Software as a Service), PaaS (Platform as a Service) und IaaS (Infrastructure as a Service). Dadurch ergibt sich für Nutzer die Vorteile kein eigenes Datencenter zu benötigen, keine langzeit Ressourcenplanung und können das Pay-as-you-go Modell verwenden und sich auf ihre Kern-Buisness Kompetenzen konzentrieren. Für Cloudanbieter ergeben sich die Vorteile von vorhandenem Investemnt zu provitieren, „Economy Of Scale“ (Mehr Ressourcen sind pro Einheit günstiger) und ihren Franchise verteidigen.

Cloudverkäufer können Cloudnutzer oder Cloudanbieter sein.

Für die Cloud wird Shared-Data verwendet, da es mehr Elastizität und Fehlerredundanz bietet, außerdem ist die Skalierbarkeit besser als bei Shared-Nothing.

Snowflake erlaubt es Nuitzer mehrere Computecluster zu deployn (virtual Warehouses, VWH), ein VWH kann verschiedene vordefinierte Größen haben. Die Queries können dann an verschiedene Cluster weitergeleitet werden und Cluster können einfach hoch und runterskaliert werden.

In der Cloud werden Tabellen in Micro-Partitions (16MB) geteilt und Metadaten werden pro Datei separat gespeichert. Die Metadaten werden genutzt, damit nur die relevanten Dateien gelesen werden (Pruning via Metadata). Dadurch kann der Scan auf einer großen Tabelle stark beschleunigt werden.

# Concurrency Control

Die gleichzeitige Ausführung in Datenbanksystemen bezieht sich auf die Fähigkeit eines Systems, mehrere Transaktionen zur gleichen Zeit auszuführen. Dies wird aus mehreren Gründen genutzt:

- Effizienzsteigerung: Durch die gleichzeitige Ausführung können Ressourcen wie CPU, Speicher und I/O effizienter genutzt werden, da Wartezeiten minimiert und Durchsatz maximiert werden.
- Verbesserte Benutzererfahrung: In Multi-User-Umgebungen ermöglicht die gleichzeitige Ausführung, dass viele Benutzer ohne signifikante Verzögerungen auf die Datenbank zugreifen können.

**Lost Update Anomaly:** Dieses Problem tritt auf, wenn zwei oder mehr Transaktionen gleichzeitig denselben Datensatz lesen und dann basierend auf dem gelesenen Wert aktualisieren. Eine Transaktion kann die Änderungen der anderen überschreiben, ohne diese zu berücksichtigen, was dazu führt, dass einige Aktualisierungen verloren gehen.

**Dirty Read Anomaly:** Eine Dirty Read Anomaly tritt auf, wenn eine Transaktion, die von einer anderen Transaktion noch nicht festgeschriebenen Daten liest. Wenn die zweite Transaktion anschließend abgebrochen wird und ihre Änderungen zurücksetzt, hat die erste Transaktion bereits auf Basis ungültiger Daten gelesen.

## Serializability

In der Theorie der Datenbanken bezieht sich Serialisierbarkeit auf die Eigenschaft einer Abfolge (oder "History") von Datenbanktransaktionen, die besagt, dass das Ergebnis der gleichzeitigen Ausführung dieser Transaktionen gleichwertig ist zu einem Ergebnis, das durch eine sequenzielle Ausführung derselben Transaktionen in irgendeiner Reihenfolge erzielt werden könnte. Die Reihenfolge der Operationen in einer Historie sind nur für conflicting Operationen relevant.

- $r_i(A)$ und $r_j(A)$ Lesen beide dasselbe Objekt, kein Konflikt also ist Reihenfolge irrelevant
- Wenn mindestens eine der Beiden Tx's ein Write ist, ist die Reihenfolge wichtig
    - $r_i(A)$ und $w_j(A)$: $T_i$ liest A und $T_j$ schreibt A
    - $w_i(A)$ und $r_j(A)$: Analog
    - $w_i(A)$ und $w_j(A)$: Analog

**Serializable Histories:** Eine History ist serialisierbar, wenn sie zu einer seriellen History äquivalent ist, in der keine zwei Transaktionen gleichzeitig ablaufen. Dies bedeutet, dass die gleichzeitige Ausführung der Transaktionen das gleiche Ergebnis liefern würde wie eine sequenzielle Ausführung ohne Überlappungen. Serialisierbare Histories sind frei von Nebenläufigkeitsanomalien und gewährleisten die Konsistenz der Datenbank.

**Non-Serializable Histories:** Nicht-serialisierbare Histories sind solche, bei denen die gleichzeitige Ausführung von Transaktionen zu Ergebnissen führen kann, die nicht durch irgendeine sequenzielle Ausführung der Transaktionen erreicht werden können. Solche Histories können zu Inkonsistenzen

führen und verschiedene Arten von Anomalien wie Lost Updates, Dirty Reads und Phantom Reads enthalten.

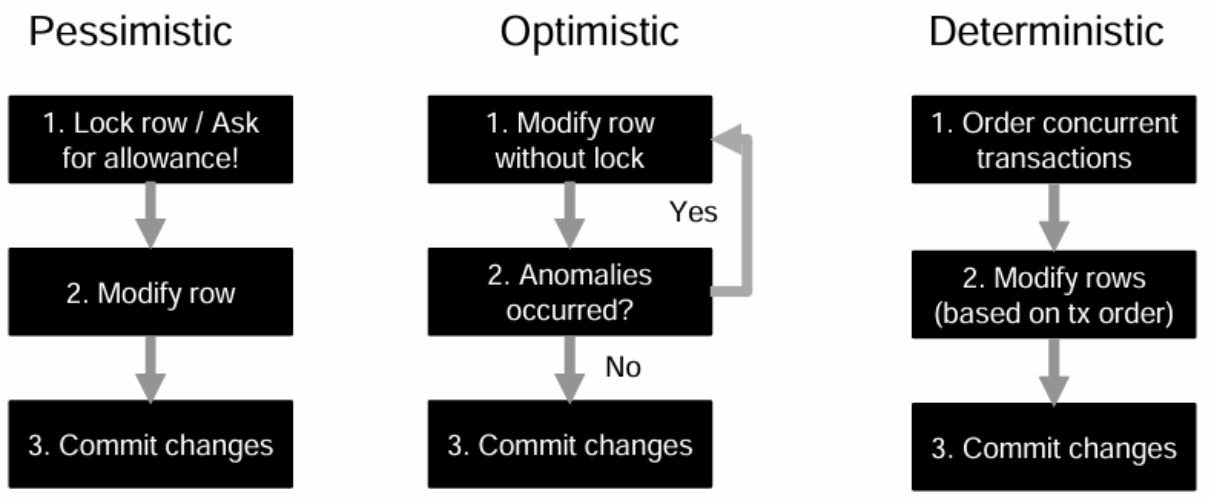
Ein **Konfliktgraph** ist ein gerichteter Graph, der verwendet wird, um die Beziehungen zwischen Transaktionen in einer History zu analysieren. Die Knoten des Graphen repräsentieren Transaktionen, und eine gerichtete Kante von Transaktion T1 zu Transaktion T2 bedeutet, dass es einen Konflikt zwischen T1 und T2 gibt, der eine Reihenfolge erfordert, in der T1 vor T2 abgeschlossen werden muss, um die Konsistenz der Datenbank zu gewährleisten. Eine History ist genau dann serialisierbar, wenn ihr Konfliktgraph keine Zyklen aufweist, was bedeutet, dass es möglich ist, eine sequenzielle Reihenfolge der Transaktionen zu finden, die mit der gleichzeitigen Ausführung konsistent ist.

| Eine Historie H ist serialisierbar, wenn der Konfliktgraph CG(H) azyklisch ist! |
|---|

## CC-Schemata

In der Theorie der Datenbanken werden Methoden zur Nebenläufigkeitskontrolle (Concurrency Control, CC) oft in drei Hauptklassen eingeteilt, um sicherzustellen, dass Datenbanktransaktionen korrekt und effizient in einer Umgebung mit gleichzeitigen Zugriffen ausgeführt werden können. Diese Klassen sind:

1. Pessimistisch: Geht davon aus, dass Konflikte zwischen Transaktionen wahrscheinlich sind und verhindert diese proaktiv durch Sperrmechanismen und andere Synchronisierungstechniken.
2. Optimistisch: Geht davon aus, dass Konflikte zwischen Transaktionen selten sind und erlaubt Transaktionen, voranzuschreiten, ohne sofortige Synchronisierung, mit der Möglichkeit, Transaktionen zurückzusetzen, wenn am Ende Konflikte entdeckt werden.
3. Deterministisch: Verwendet Zeitstempel, um die chronologische Ordnung von Transaktionen zu bestimmen und Konflikte basierend auf dieser Ordnung zu lösen, ohne direkte Sperren auf Datenobjekte anzuwenden.




## Lock-based Concurrency Control

Lock-based Concurrency Control verwendet Sperren, um den Zugriff auf Datenbankobjekte zu kontrollieren und sicherzustellen, dass Transaktionen ohne Interferenzen von anderen Transaktionen durchgeführt werden können. Two-Phase Locking (2PL) ist ein Protokoll, das sicherstellt, dass eine Transaktion in zwei Phasen abläuft:

1. Locking Phase: In dieser Phase erwirbt eine Transaktion alle erforderlichen Sperren, bevor sie Änderungen durchführt. Es werden keine Sperren freigegeben.

2. Unlocking Phase: Nachdem alle Operationen der Transaktion abgeschlossen sind, werden alle gehaltenen Sperren freigegeben. In dieser Phase werden keine neuen Sperren erworben.

Die Implementierung von 2PL erfordert eine Sperrtabelle (implementiert als Hash-Tabelle (Schlüssel der Hash-Tabelle = PK einer Zeile, Wert = Warteschlange für Sperren)). Diese Tabelle verfolgt, welche Transaktion Lock für welche Zeile hält, sowie ausstehende Sperranforderungen. Eine neue Anforderung wird an das Ende der Warteschlange angehängt und freigegeben, wenn keine andere, nicht kompatible freigegebene Sperre in der Warteschlange ist. Entsperrungsanforderungen führen dazu, dass die Anforderung gelöscht wird und neue Locks freigegeben werden. Wenn die Transaktion aborts / commits, werden alle wartenden oder freigegebenen Anforderungen der Transaktion gelöscht.

**Cascading Aborts** treten auf, wenn das Zurücksetzen (Abort) einer Transaktion das Zurücksetzen einer oder mehrerer anderer Transaktionen erfordert, die von den Daten gelesen haben, die von der ursprünglichen Transaktion geändert wurden. Dies kann zu einer Kaskade von Abbrüchen führen, die mehrere Transaktionen betreffen.

**Strict Two-Phase Locking (Strict 2PL)** ist eine Variante von 2PL, bei der alle Sperren bis zum Ende der Transaktion gehalten werden, einschließlich nach dem Commit-Punkt. Dies verhindert Cascading Aborts, da keine Änderungen für andere Transaktionen sichtbar gemacht werden, bevor die Transaktion erfolgreich abgeschlossen ist.

## Deadlocks

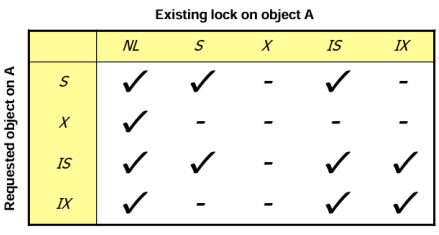
**Deadlock Detection** bezieht sich auf das Verfahren zur Erkennung von Deadlocks, Situationen, in denen zwei oder mehr Transaktionen auf Ressourcen warten, die von der jeweils anderen Transaktion gesperrt sind, was zu einem Stillstand führt. Deadlock-Erkennungsalgorithmen, wie das Warten-auf-Graph-Verfahren, identifizieren Zyklen im Graphen der wartenden Transaktionen, um Deadlocks zu erkennen.

**Wait-Die** ist eine Deadlock-Präventionsstrategie, die Zeitstempel nutzt, um zu entscheiden, ob eine Transaktion warten soll oder abgebrochen wird, wenn sie auf eine von einer anderen Transaktion gehaltene Ressource zugreifen möchte. Ältere Transaktionen dürfen warten, während jüngere Transaktionen in Konfliktsituationen abgebrochen und neu gestartet werden.

**Preclaiming** verhindert Deadlocks, indem eine Transaktion alle erforderlichen Ressourcen anfordert und erhält, bevor sie ihre Operationen beginnt. Kann sie nicht alle Sperren sofort erhalten, gibt sie alle gehaltenen Sperren frei und versucht es später erneut, was Deadlocks effektiv vermeidet.

## Multiple Granularity Locking

**Multiple Granularity Locking (MGL)** ist ein Sperrenkonzept, das es ermöglicht, Sperren auf verschiedenen Ebenen der Datenhierarchie zu setzen, z.B. auf der Ebene der gesamten Datenbank, von Tabellen oder einzelnen Zeilen. MGL ermöglicht eine flexiblere und effizientere Sperrenverwaltung, indem es anpasst, wie grob oder fein die Sperren angewendet werden, basierend auf den Anforderungen der Transaktion.




Neben den traditionellen Sperrenmodi (wie S für Shared und X für Exclusive) führt MGL erweiterte Sperrenmodi ein, um die Verwaltung von Sperren auf unterschiedlichen Granularitätsebenen zu unterstützen. Zu diesen erweiterten Modi gehören:

- Intention Shared (IS): Zeigt an, dass eine Transaktion beabsichtigt, Shared-Sperren auf feineren Granularitätsebenen innerhalb der gesperrten Einheit zu setzen.

- Intention Exclusive (IX): Zeigt an, dass eine Transaktion beabsichtigt, Exclusive- oder Shared-Sperren auf feineren Granularitätsebenen zu setzen.
- Shared Intention Exclusive (SIX): Kombination aus Shared- und Intention Exclusive-Sperre, die angibt, dass eine Transaktion eine Shared-Sperre auf der aktuellen Ebene und beabsichtigt, Exclusive-Sperren auf niedrigeren Ebenen zu setzen.

Lock Protocol:

1. Bevor ein Objekt in S oder IS gesperrt werden kann, müssen alle Vorgänger in der Historie durch derselben Transaktion im IS-Modus durch den TX gesperrt sein
2. Bevor ein Objekt in X oder IX gesperrt werden kann, müssen alle Vorgänger in der Histtorie von derselben Transaktion im IX-Modus gesperrt sein
3. Die Locks werden Bottom-Up freigegeben, sodass keine Transaktion ein Lock eines Nachfolgers hält

### Multi-Versoning Concurrency Control

Ein typisches Problem von Lock-based Schemata sind, dass die Leser die Schreiber blockieren. MVCC behält mehrere Versionen derselben Datenzeile und verhindert so blockieren. Jeder erfolgreiche Write erstellt eine neue Version einer Zeile. Wenn eine Lese-Operation angekündigt wird, wird die passende Version der Zeile ausgewählt basierend auf dem Start der Transaktion, welche die Operation angefragt hat.

#### Snapshot Isolation

Snapshot Isolation ist ein typisches MVCC Schema.

Eine Transaktion T wird mit Snapshot Isolation ausgeführt

- Reads:
  - TX liest Version der Tupel wie am Start von T
  - Updates von anderen gleichzeitigen Tx's sind von T nicht sichtbar
- Writes:
  - First-commiter-wins-Regel: T comitted nur, wenn keine anderen gleichzeitigen Transaktionen Daten geschrieben haben, die T schreiben möchte
  - Kind-of-Optimistic Ausführung

SI verhindert mehrere Anomalien:

- No dirty read: d.h. kein Lesen von unbestätigten Daten
- No lost update: d.h. Aktualisierungen, die von einer Transaktion vorgenommen werden, werden von einer anderen Transaktion überschrieben, die die Aktualisierung nicht gesehen hat
- No non-repeatable read: d.h. wenn ein Lesevorgang erneut ausgeführt wird, sieht er denselben Wert

Aber es entstehen auch neue Anomalien: **Write Skew**: Eine Situation, in der zwei Transaktionen auf zwei Datensätze zugreifen, diese aber in einer Weise ändern, die in der Gesamtheit nicht konsistent ist. Beide Transaktionen können ihre Änderungen erfolgreich durchführen, da sie keine direkten Konflikte auf einem einzelnen Datensatz haben, aber das Endergebnis kann inkonsistent sein.

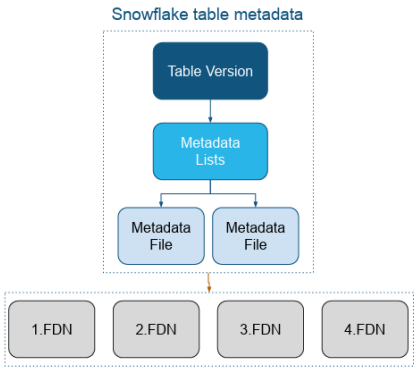
## Snowflake

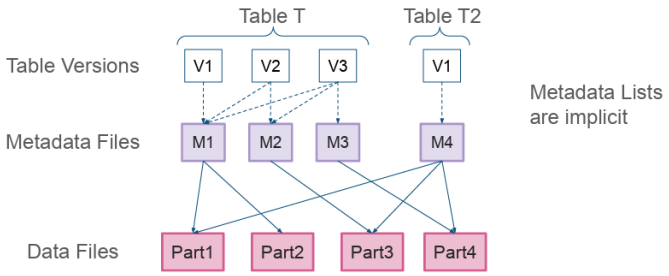
Snowflake ist ein cloud-basiertes Data-Warehouse, das für seine einzigartige Architektur bekannt ist, die eine hohe Skalierbarkeit und Leistung bietet. Es trennt Speicher und Rechenressourcen, sodass

Nutzer die Rechenleistung dynamisch an ihre Anforderungen anpassen können, ohne den zugrundeliegenden Speicher zu beeinflussen.

## Snwoflake Table Structure

1. Micro-Partitions: Daten in Snowflake-Tabellen werden automatisch in Micro-Partitions unterteilt, die effizient komprimierte und verschlüsselte Speichereinheiten sind, typischerweise zwischen 50 MB und 500 MB groß. Diese Feingranularität ermöglicht effizientes Pruning und Datenzugriff.
2. Columnar Storage: Snowflake speichert Daten in einer spaltenbasierten Struktur, wobei jede Micro-Partition die Daten spaltenweise speichert. Dies verbessert die Abfrageleistung, insbesondere für OLAP-Workloads, da nur die benötigten Spalten gelesen werden müssen.
3. Immutable Data: Sobald Daten in Snowflake geschrieben werden, sind sie unveränderlich. Änderungen an den Daten führen zur Erstellung neuer Micro-Partitions, während die alten für Time-Travel-Funktionalitäten beibehalten werden.




**DMLs** sind eine Gruppe von Sprachbefehlen in SQL, die verwendet werden, um Daten in einer Datenbank zu manipulieren. Dazu gehören Befehle wie INSERT, UPDATE, DELETE und SELECT, die es Nutzern ermöglichen, Daten hinzuzufügen, zu ändern, zu löschen und abzufragen.

**Time Travel** in Snowflake ermöglicht es Benutzern, Daten in der Vergangenheit zu betrachten und zu interagieren, indem sie auf frühere Versionen von Daten innerhalb eines konfigurierbaren Zeitraums (bis zu 90 Tage) zugreifen. Dies ist nützlich für die Wiederherstellung von Daten nach versehentlichen Löschungen oder Änderungen und für die Analyse historischer Daten.

**Zero-Copy Cloning** in Snowflake ermöglicht das Erstellen von Klonen (Duplikaten) von Datenbanken, Schemata oder Tabellen, ohne dass zusätzlicher Speicherplatz für die geklonten Daten benötigt wird. Änderungen an den Originaldaten oder den Klonen beeinflussen sich nicht gegenseitig, was eine effiziente Datenverwaltung und -testung ermöglicht.

## Pruning

In Snowflake wird Pruning verwendet, um die Effizienz von Datenabfragen zu verbessern, indem unnötige Datenblöcke, bekannt als Micro-Partitions, von der Abfrageverarbeitung ausgeschlossen werden. Pruning kann auf zwei Ebenen stattfinden: während der Kompilierung (Compile-Time) und zur Laufzeit (Run-Time).

**Compile-Time Pruning**: Beim Compile-Time Pruning nutzt Snowflake die Metadaten der Tabelle, um bereits vor der eigentlichen Ausführung der Abfrage zu bestimmen, welche Micro-Partitions nicht für die Ergebnisse relevant sind. Diese Entscheidung basiert auf den in den Metadaten gespeicherten Statistiken wie Min-/Max-Werten der Spalten.

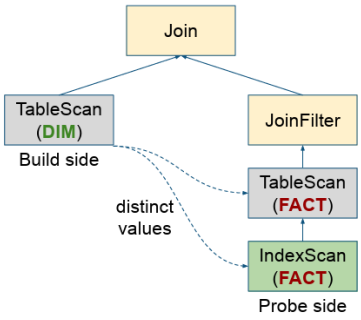
**Run-Time Pruning**: Run-Time Pruning tritt während der Ausführung der Abfrage auf und ist besonders nützlich für Abfragen, die semi-strukturierte Daten wie JSON, Avro oder Parquet

verarbeiten. Snowflake analysiert die Abfragebedingungen in Echtzeit und bestimmt, welche Micro-Partitions relevante Daten enthalten könnten. Durch das Ausschließen irrelevanter Micro-Partitions kann Snowflake die I/O-Operationen reduzieren und die Abfrageleistung verbessern.

## Indexing

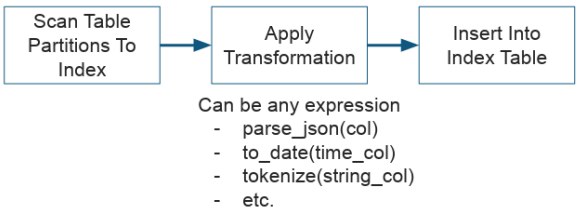
Im traditionellen Datenbankkontext verwenden Systeme Indizes, um den Zugriff auf Daten zu beschleunigen. Snowflake verfolgt jedoch einen anderen Ansatz und setzt stattdessen auf automatisches Micro-Partitioning und Metadaten-basiertes Pruning. Anstatt manuell erstellte Indizes zu verwenden, nutzt Snowflake die in den Micro-Partitions enthaltenen Metadaten, um effizientes Pruning durchzuführen und die Datenzugriffszeiten zu minimieren. Dieser Ansatz reduziert die Notwendigkeit für traditionelle Indexstrukturen und vereinfacht die Datenverwaltung und -optimierung in der Cloud-Umgebung.

## Search Optimization Service



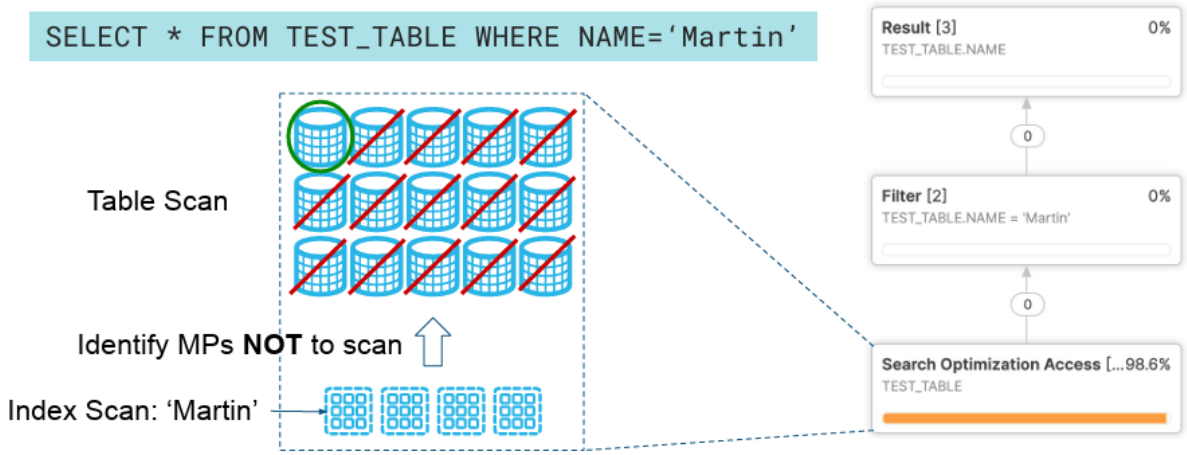

Der Search Optimization Service von Snowflake verwaltet automatisch pruning indices basierten auf einem Bloom Filter (speicherplatz-effiziente probabilistische Datenstruktur, die zum Testen verwendet wird, ob ein Element Teil einer Menge ist). Beschleunigt Gleicheit, Substring, Semi-Structured und Geo Datensuchen. Mit einer gegebenen Konstante, alle Partitionen entfernen, die diese nicht enthalten. Search Optimierung wird verwednet, wenn die Query Predikate SO unterstützen und der der Optimierer eine Verbesserung erwartet. SO wird verwendet, wenn die Größe der zu secannenden SO Daten deutlich kleiner ist als die Größe der Table Data welche gescannt werden soll. SO kann Gleichheits Joins beschleunigen, vorallem diese, die eine kleinen Build Side Größe haben und der Probe Table nicht auf dem Join Key geclustered wurde.

SO zu verwenden ist eine dynamische Entscheidung basierend auf der Anzahl unterscheidbarer Werte. Wenn die Daten diskretierbar sind, können sie geindext werden. Nutzer können auswählen, welche Spalten geindext werden und für welche Suchoperation oder Snowflake entscheiden lassen.




```
ALTER TABLE X ADD SEARCH OPTIMIZATION
[ON (<SEARCH_METHOD_WITH_TARGETS>)+];
```

**TARGET** = column, variant path, *
**METHOD** = EQUALITY, SUBSTRING, GEO



## Materialization und Reuse

**Materialization** in Snowflake bezieht sich typischerweise auf die Erstellung von physischen Datenstrukturen oder Zwischenergebnissen, die aus Abfragen resultieren, und ihre Speicherung für zukünftige Verwendung. Dies kann beispielsweise das Ergebnis einer komplexen Abfrage sein, das in einer temporären Tabelle oder als "materialized view" gespeichert wird. **Materialized Views** sind eine besondere Art von Datenstruktur in Snowflake, die das Ergebnis einer Abfrage speichert und automatisch aktualisiert, um Änderungen in den zugrundeliegenden Basistabellen widerzuspiegeln. Die Verwendung von Materialized Views kann die Leistung verbessern, indem teure Berechnungen vermieden werden, wenn ähnliche oder identische Abfragen in der Zukunft ausgeführt werden.

**Reuse** in Snowflake bezieht sich auf die Fähigkeit des Systems, Ergebnisse oder Datenstrukturen, die bereits berechnet oder materialisiert wurden, effizient wiederzuverwenden, um ähnliche Abfragen schneller auszuführen. Dazu gehört Result Set Caching, Zero-Copy Cloning und Time Travel.

Wenn weder Daten entfernt oder wiederverwendet werden können muss man effizient sein:

- Effizient vom Cloudspeicher lesen (IO Größe und Anzahl)
- Aus dem lokalen und cloud-Cache lesen
- Nur die Spalten laden, welche benötigt werden
- Verbessertes Dateiformat
- Kompression

## Automatisches Clustering

Natürlicherweise folgt das Datenlayout (also clustering) der Reihenfolge der DML-Operationen. Idealerweise sollte das Datenlayout aber dem relevantesten Attribut folgen. Das Optimieren des Datenlayouts garantiert konsistente Query-Performanz.

Der Kunde stellt Clustering Keys bereit, automatische serverlose Clusterverwaltung passiert im Hintergrund. Bei Snowflake bedeutet das, eine annähernde Sortierfolge zu verwalten. Partitionen werden reorganisiert, um ähnliche Cluster-Keyranges zu beinhalten. Das Ziel ist nicht zu sortieren aber eine gute min/max Pruning Performanz zu erreichen.

- Recluster inline mit den Veränderungen: Die gesamte Last wird auf die DML-Operationen gelegt. Direkte Auswirkung auf die Kundenperformanz.
- Periodisches Batch reclustering: Teuer, blockiert andere Änderungen und ist unpassend um in den Hintergrund laufen zu können. Die Queryperformanz wird immer schlechter bis zum nächsten Recluster. Nicht praktisch für Petabyte große Tabelle.
- Inkrementelles Reclustering im Hintergrund: Hintergrundtask wählt periodisch einen kleinen Satz and Kandidatdatein, welche reclustered werden um das optimale Tabellenlayout beizubehalten.

### Clustering Metriken

Die **Clustering Width** bezieht sich auf die Anzahl der Spalten oder Attribute, die für das Clustering einer Tabelle oder eines Datensatzes verwendet werden. In einem Clustering-Kontext bestimmt die Breite, wie Daten auf Basis bestimmter Schlüssel oder Attribute gruppiert werden. Eine breitere Clustering-Konfiguration kann mehr Attribute für das Clustering umfassen, was zu einer feineren Segmentierung der Daten führt. Dies kann die Abfrageleistung für Workloads verbessern, die speziell auf diese Attribute abzielen, kann aber auch die Komplexität und den Overhead für das Datenmanagement erhöhen.

Die **Clustering Depth** bezieht sich auf die Anzahl der Ebenen oder Schichten der Clusterhierarchie innerhalb der Datenorganisation. Eine tiefere Clustering-Struktur bedeutet, dass Daten in mehreren

verschachtelten Ebenen organisiert sind, was zu einer detaillierteren und möglicherweise effizienteren Organisation führen kann, insbesondere wenn die Abfragen stark auf die Struktur der Daten abgestimmt sind. Tiefe Clustering-Strukturen können jedoch auch die Komplexität erhöhen und die Performance beeinträchtigen, wenn die Abfragen nicht gut auf die Clustering-Hierarchie abgestimmt sind.

**Partition Selection** bezieht sich auf den Prozess der Auswahl spezifischer Partitionen oder Segmente der Daten für die Verarbeitung einer Abfrage. In Systemen, die Daten partitionieren (wie Snowflake, das Micro-Partitions verwendet), kann die effiziente Auswahl von Partitionen, die für eine Abfrage relevant sind, die Anzahl der zu lesenden Daten erheblich reduzieren und somit die Abfrageleistung verbessern. Dies wird oft durch Pruning-Techniken erreicht, bei denen Metadaten wie Min-/Max-Werte der Partitionen verwendet werden, um schnell zu bestimmen, welche Partitionen relevante Daten enthalten könnten und welche ausgeschlossen werden können.

## MapReduce

MapReduce ist ein Programmiermodell und eine zugehörige Implementierung zur Verarbeitung und Generierung großer Datensätze auf einem Cluster aus Computern. Entwickelt von Google, ist MapReduce besonders nützlich für Aufgaben, bei denen große Mengen an rohen Daten in nützlichere Formate transformiert werden müssen. Das Modell ist in zwei Hauptphasen unterteilt: Map und Reduce.

**Map-Phase:**

- Input: Die Map-Phase beginnt mit dem Eingang von Key-Value-Paaren. Die Eingabedaten werden typischerweise aus einem verteilten Dateisystem gelesen und in kleine Stücke oder "Splits" aufgeteilt, die parallel verarbeitet werden können.
- Mapping-Funktion: Jedes Input-Paar wird durch eine Mapping-Funktion verarbeitet, die vom Benutzer definiert wird. Diese Funktion transformiert das Input-Paar in ein oder mehrere Zwischen-Key-Value-Paare. Beispielsweise könnte eine Mapping-Funktion, die auf Textdaten angewendet wird, jedes Wort in einem Dokument zählen und für jedes Vorkommen eines Wortes ein Key-Value-Paar mit dem Wort als Schlüssel und der Zahl 1 als Wert erzeugen.

**Shuffle- und Sort-Phase:**

- Nach der Map-Phase sammelt und sortiert das System automatisch die Zwischen-Key-Value-Paare nach den Schlüsseln. Diese Phase wird manchmal auch als "Shuffle" bezeichnet. Daten mit demselben Schlüssel werden zusammengeführt, sodass sie in der Reduce-Phase effizient verarbeitet werden können.

**Reduce-Phase:**

- Input: Die Eingabe für die Reduce-Phase sind die gruppierten Zwischen-Key-Value-Paare von der Map-Phase.
- Reduce-Funktion: Für jeden einzigartigen Schlüssel und seine Liste von Werten führt das System eine Reduce-Funktion aus, die ebenfalls vom Benutzer definiert wird. Diese Funktion verarbeitet die Werte auf eine Weise, die für die Anwendung sinnvoll ist, wie das Summieren von Zählungen im Wortzählungsbeispiel. Das Ergebnis sind Output-Key-Value-Paare, die das Endergebnis der Verarbeitung darstellen.

### Apache Hadoop

Hadoop ist ein Open-Source-Framework, das für die Speicherung und Verarbeitung großer Datenmengen (Big Data) in einem verteilten Computing-Umfeld entwickelt wurde. Es wurde von der

Apache Software Foundation entwickelt und basiert auf dem MapReduce-Algorithmus von Google sowie dem Google File System (GFS) Papier. Hadoop ermöglicht es, Anwendungen auf Systemen mit Tausenden von Knoten (Computern) und Petabytes an Daten zu betreiben.

Kernkomponenten von Hadoop:

- **Hadoop Distributed File System (HDFS):** Ein verteiltes Dateisystem, das Daten über mehrere Maschinen speichert, um hohe Aggregatbandbreite und Zuverlässigkeit zu bieten. HDFS teilt Dateien in Blöcke (standardmäßig 128 MB oder 256 MB groß) und verteilt diese Blöcke auf verschiedene Knoten im Cluster, wobei Kopien (Replikate) zur Gewährleistung der Fehlertoleranz erstellt werden.
- **MapReduce:** Ein Programmiermodell und eine Implementierung für die Verarbeitung und Generierung großer Datensätze. Anwendungen, die MapReduce verwenden, sind in zwei Phasen unterteilt: Die Map-Phase, die ein Paar "Key-Value" liest und ein Zwischenergebnis generiert, und die Reduce-Phase, die die Ausgabe der Map-Phase als Eingabe nimmt und die endgültigen Ergebnisse kombiniert.

Wie Hadoop funktioniert:

- **Job- und Ressourcenmanagement:** Hadoop verwendet YARN (Yet Another Resource Negotiator) für die Cluster-Ressourcenverwaltung, die die Aufgabe hat, Rechenressourcen zu verwalten und Benutzeranwendungen (Jobs) zu planen.
- **Datenverarbeitung:** Ein typischer Hadoop-Verarbeitungszyklus beginnt mit der Eingabe von Daten in das HDFS. Ein MapReduce-Job wird dann gestartet, der die Daten verarbeitet. Die Map-Funktion führt die Datenverarbeitung in kleineren Teilen parallel auf verschiedenen Knoten aus, während die Reduce-Funktion die Ergebnisse der Map-Funktionen zusammenführt, um das Endergebnis zu generieren.
- **Fehlertoleranz:** Hadoop ist hochgradig fehlertolerant, da es Daten automatisch repliziert und die Verarbeitung auf einem anderen Knoten neu startet, falls ein Knoten ausfällt.

### Apache Spark

Hadoop schreibt jedes MR-Ergebnis auf das HDFS, die Lösung dazu bietet Apache Spark. Spark verwendet In-Memory Data, was 10-100x so schnell ist. Spark ist der Nachfolger von Hadoop

## Streaming

**Definition Continuous Queries**: A new class of queries, continuous queries, are similar to conventional database queries, except that they are issued once and henceforth run “continually” over the database.

Anstelle von DBMS werden mehr DSMS (Data Stream management Systems) verwendet.

| DBMS | DSMS |
|---|---|
| Persistente Daten | Transient Daten |
| One-time Queries (pull based) | Kontinuierliche Queries (Push based) |
| Zufällige Zugriffe | Sequentieller Zugriff |
| Idealer Ablaufplan kann für jede Query festgelegt werden | Ablaufplan muss an eventuell ändernde Datenraten angepasst werden |
| Tabellen | Streams |
| Tabellen sind mehr oder weniger statische Daten | Streams sind kontinuierliche Daten |

## Stream

Ein Stream ist eine nichtbegrenzte Tabelle, konzeptionell unendlicher Satz and Dateneinträgen / Events, welche kontinuierlich über Zeit produziert werden. Ist nicht vollständig vorhanden, bevor die Verarbeitung beginnt.

Streams arbeiten mit einem **Push Modell**, die Datenproduktion und Verarbeitung wird von der Quelle kontrolliert, häufig wird ein Publish/Subscribe Modell verwendet.

Die **Zeit** in Streams spielt eine zentrale Rolle, wichtig sind **Eventzeit** (Datenitem Erstellzeit), **Einfügezeit** (Ingestion time, Systemzeit, wenn Daten empfangen werden) und **Verarbeitungszeit** (Processing Time, Systemzeit, wenn Daten verarbeitet werden). Häufig muss begründet werden wann Daten produziert und verarbeitet werden.

**Definition**: Stream $S = s_1, s_2, \dots, s_i, s_{i+1}, \dots, s_i =$<timestamp, data item>

Der **timestamp** ist meistens die Eventzeit und das Daten Item ein Key-Value-Pair. Das DSMS hat keinen Einfluss darauf, wann und in welcher Reihenfolge die Daten ankommen und weiß nicht, wie lang der Datensream sein wird.

## Stream Processing

Der kontinuierliche Datenstream wird mit stehenden, überwachenden Queries verarbeitet und liefert so ein kontinuierlichen Ergebnisstream.

**Anforderungen**:

1. Daten verarbeiten, ohne sie zu speichern
2. Query verwendet high-level Sprache
3. Umgang mit verspäteten, verlorenen oder out-of-order Daten
4. Generierung von deterministischen und korrekten Ergebnissen
5. Integration von Batch und Stream Processing
6. Garantieren von Integrität und Verfügbarkeit der Daten
7. Automatisches Partitioning und Skalierung
8. Verarbeiten und antworten „sofort“ (mit wenig Latenz)

**Stateless Operatoren**: Werde auf direkt auf jedes Item angewendet und behalten keinen State

- Konvertierung (zB. Fahrenheit zu Celsius)
- Filter (zB. Temperatur > 60°C)

**Statefull Operatoren**: Muss den Input Status behalten um den Output zu berechen.

- Aggregation (zB. AVG Temperatur)

## Stream Windows

DSMS können Datenstreams in Windows verarbeiten, dabei wartet das System für x Zeit oder Items und gibt dann ein Ergebnis pro Window zurück. So kann zB der Durchschnitt immer auf einem Window berechnet werden. Datenstreams können auch Window-weise gejoint werden.

Stream Windows haben eine Größe und einen Slide, der Slide beschreibt, den Abstand zum letzten Window-beginn.

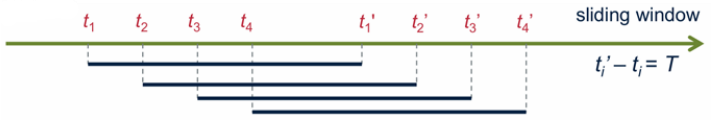
**Window-Arten:**

- Zeitbasierte Windows: Größe ist festgelegt durch die Tupel, welche in einem bestimmten Zeitfenster ankommen.

- Tupel-basierte Windows: Größe ist durch die Anzahl der Tupel festgelegt.
- Semantische Windows: Fenstergröße und Inhalt sind durch den Tupelinhalt festgelegt.

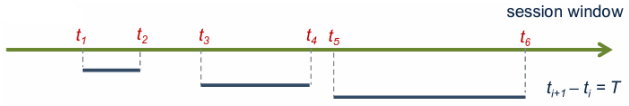
**Sliding-Arten:**

- Sliding Window: bewegt sich automatisch weiter, wenn neue Tupel ankommen (Overlap möglich)

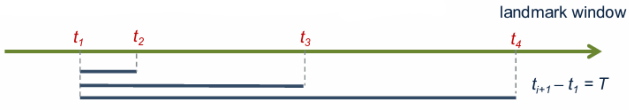



- Tumbling Window: wartet, bis das neue Window voll ist (kein Overlap)


- Session Window: schließt, wenn keine Aktivität also kein neuen Tupel mehr für eine gewisse Zeit



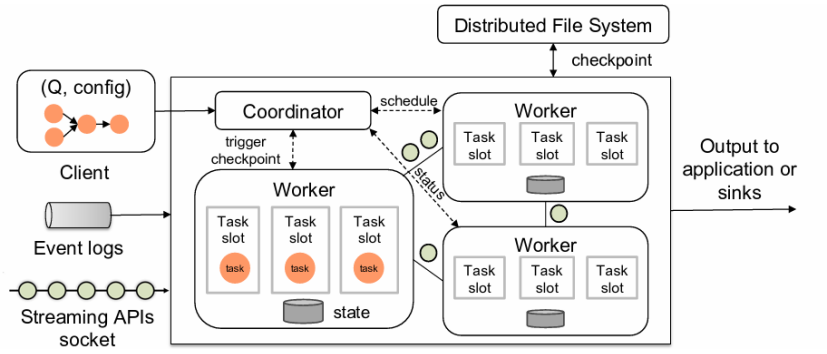

- Landmark Window: festgelegte Startzeit, Endzeit wird immer für neue Tupel erweitert



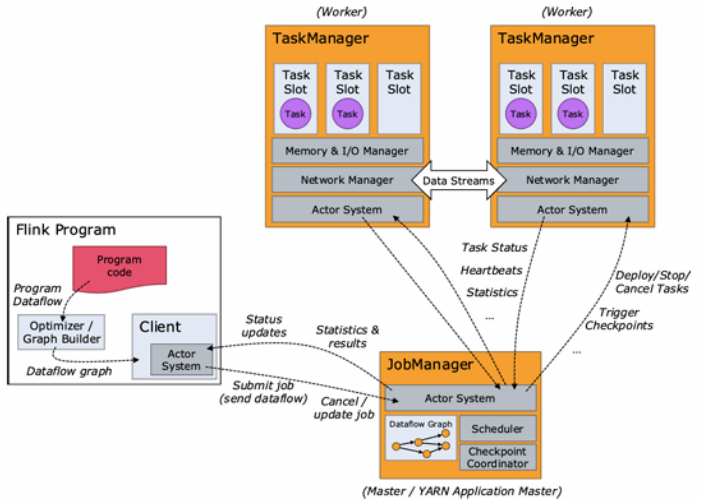

## Apache Flink

Apache Flink implementiert ein natives Streaming Model (kein MR). Durch die Verarbeitung von Streams als Pipelines kann geringere Latenz erzielt werden. Ähnlich wie Pipelines in DBMS aber Operatoren pushen Daten weiter und Daten müssen mit Buffern weitergereicht werden.

Distributed Dataflow Systems




Apache Flink Architektur




Konzepte und Komponente

**Flink Job**: Pipeline von verbundenen Operatoren, welche Daten verarbeiten

**JobGraph**: Die Operatoren formen zusammen den JobGraph (Dataflow Graph). Jeder Operator hat eine bestimme Anzahl an Subtasks, welche parallel ausgeführt werden.

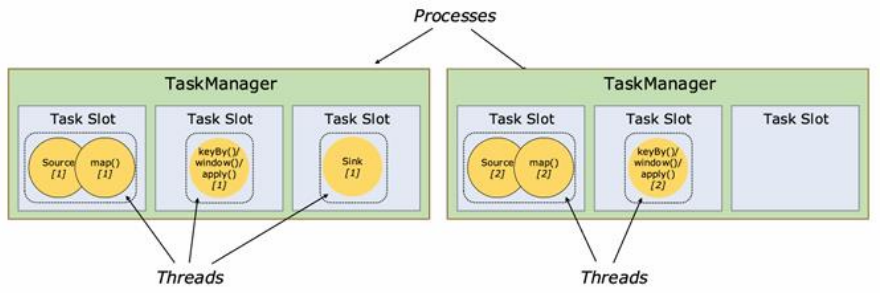
**Subtask** ist eine Ausführungseinheit in Flink. Er konsumiert die Nutzer Records von anderen Subtasks, verarbeitet diese und produziert Output Records, welche danach von anderen nachfolgenden Subtasks konsumiert werden können. Subtasks sind die Knoten des Execution Graphs.

**Flink Client:** kompiliert Streaming Anwendungen in einen Datenflow Graphen und sendet diese an den JobManager.

**Job Manager**: verantwortlich für Ressourcen de-/allocation in einem Cluster, erstellt Task Slots, versendet die Flink Anwendungen und startet einen JobMaster für jeden Job. Mehrere JobManager (mit einem Führenden) werden für hoch qualitative Setups verwendet. Der Job Manager koordiniert Checkpoints und Wiederherstellung nach Fehlern.

**Task Manager:** Auch genannt Worker, führt den Task des Datenflows aus und es gibt immer mindestens einen Task Manager. Jeder Worker ist ein JVM-Prozess, der ein oder mehrere Subtasks ausführt. Die kleinste Einheit des Task Managers ist der Task Slot, mehrere Operatoren können in einem Task Slot ausgeführt werden.

**Resource Manager**: verwaltet die Taskslots mit verschiedenen Anbietern wie Hadoop YARN, Kubernetes oder Standalone Deployments. Abhängig von externen Tools für de-/allocation der Ressourcen.

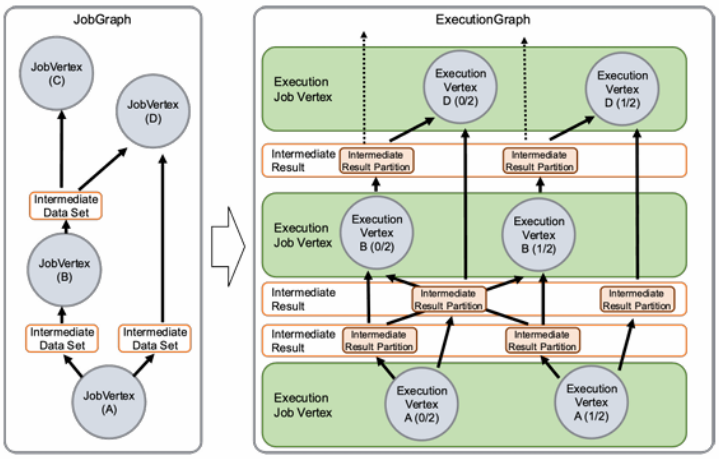



Flink kettet Operator Subtasks in **Tasks** zusammen. Jeder Task wird von einem CPU-Thread ausgeführt. Zwei aufeinanderfolgende Transformationen werden im selben Thread ausgeführt um die thread-to-thread Handover Overhead zu reduzieren.

Jeder **Task Slot** repräsentiert ein festgelegtes Subset von Ressourcen des Task Managers. Slots teilen nur den verwalteten Speicher der Tasks, es herrscht keine CPU-Isolation. Durch Anpassen der Task Slots, können Nutzer definieren, wie Subtasks isoliert werden. 1 Slot pro TM bedeutet, dass jede Task Gruppe eine eigene JVM bekommt. N Slots bedeutet, dass mehrere Subtasks dieselbe JVM teilen. Der Vorteil von Slot Sharing ist, dass Flink automatisch abhängig von der Parallelität die beste Anzahl and Task Slots berechnet. Dadurch können Ressourcen besser verwaltet werden.

**Dispatcher:** bietet ein REST-Interface, um neue Flink Anwendungen einzureichen und einen neuen JobMaster für jeden Job zu erstellen.

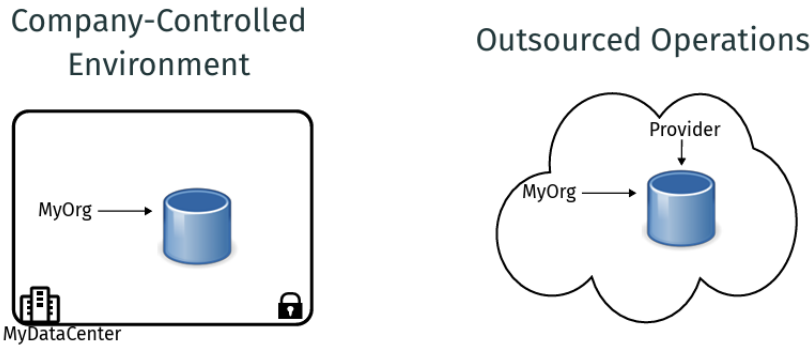
**Job Master:** Verwaltet Ausführung eines einzelnen JobGraph. Mehrere Jobs haben je ihren eigenen Job Master.




- Ziel des Jobmasters ist es die verteilten Tasks zu überwachen und zu entscheiden, wenn ein neuer Task ausgeführt werden soll
- Der JobGraph ist die Representation des Dataflow Graphs mit Parallelismus
- Der Execution Graph ist die parallele Versionen des Job Graphen

## Security und Privacy

Typischerweise werden Company-Controlled Environments verwendet, aber durch die Cloud werden immer mehr Operationen outgesourced.




Das bringt aber auch mehr Gefahren mit sich, so können zum Beispiel **Confidentiality** (Data Breaches) und **Data Integrity** (Data Manipulations) gefährdet werden.

Daten werden auch immer mehr mit anderen Partnern geteilt, was wiederrum noch mehr Datensicherheits Herausforderungen mit sich bringt. So können zum Beispiel **Sovereignty** (Limitierter Daten Zugriff) oder die **Compute Integrity** (Kontrolle über Datennutzung) gefährdet werden.

Aktuelle Techniken zum Schutz existieren, aber haben einen hohe Performanz Overhead. Wie können also die Effizienz von Sicherheit und Privatsphäre Techniken verbessert werden und wie kann man effiziente und sichere Datensysteme bauen mit diesen Techniken?

- Encryption -> Confidentiality

- Crypto. Data Structures -> Data Integrity
- Secure Hardware -> Compute Integrity, …
- Transparency Logs -> Tamper-Detection, Auditability

## Security Toolbox

Sicherheit bedeutet ein Ziel in Anwesenheit eines Gegners zu erreichen

- Policy /Guarantees (Ziele): Confidentiality, Integrity, Availability
- Threat Model (Gegner): Annahmen über Gegner
- Mechanismus: Tools um die Policies einzuhalten

**Hash Funktionen:** Variabel langer Input soll in einen kurzen und bündigen Output übergehen.

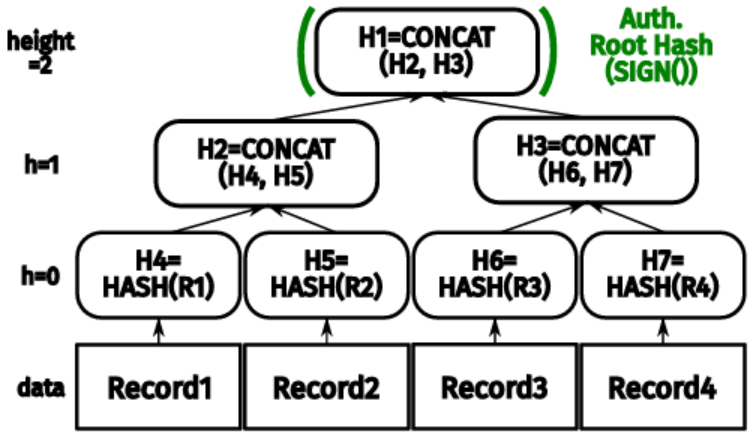
| | Checksum | Hash | Crypto. Hash |
|---|---|---|---|
| Avalanche Effect<br>$x' \rightarrow h'$ | X | X | X |
| Collision Resistant<br>$x, y: H(x) \neq H(y)$ | - | X | X |
| Pre-image Resistant<br>$y: H(y) = h$ | - | - | X |
| Second Pre-image Resistant<br>$y \neq x: H(y) = H(x)$ | - | - | X |

Digitale Signaturen:

| | Hash | MAC | HMAC | Digital Signature |
|---|---|---|---|---|
| Integrity | X | - | X | X |
| Authentication | - | X | X | X |
| Non-Repudiation | - | - | - | X |
| Key-Type | None | Symmetrisch | Symmetrisch | Asymmetrisch |

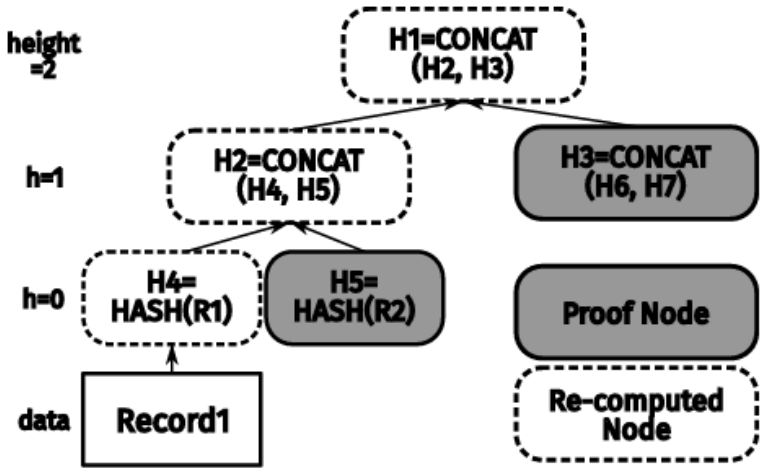
## Merkle Trees

Ein Merkle-Baum ist ein Baum, in dem jeder Blattknoten mit einem Datenblock markiert ist und jeder Nicht-Blattknoten mit einem kryptographischen Hash seiner Kindknoten markiert ist. Dies bedeutet, dass jeder Knoten im Baum den Hash eines Datenelements oder den Hash von Hashes seiner Kindknoten enthält. Die Wurzel des Baumes, bekannt als Merkle-Wurzel, enthält einen einzigen Hash, der eine Zusammenfassung der gesamten darunterliegenden Daten repräsentiert. Dadurch können mit einem einzelnen Hash eine Änderung irgendwo im System festgestellt werden.



**Proof Nodes** sind essenziell, um einen Merkle Proof (Merkle-Beweis) zu liefern, der die Existenz und Integrität eines spezifischen Datenelements innerhalb des gesamten Datensatzes nachweist, ohne den gesamten Datensatz herunterladen zu müssen. Für ein gegebenes Datenelement beinhaltet ein Merkle Proof:

1. **Das Ziel-Datenelement:** Das spezifische Datenelement, für das der Beweis erbracht wird.
2. **Die Proof Nodes:** Eine Reihe von Knoten (Hashes), die vom Ziel-Datenelement bis zur Wurzel des Baumes führen.
3. **Die Sibling Nodes:** Die Geschwisterknoten (die Knoten auf derselben Ebene, die nicht direkt im Beweispfad liegen) sind notwendig, um den Hash bis zur Wurzel zu berechnen und den Beweis zu validieren.




Die Hash Operationen sind teure Operationen und können nur auf einem Thread ausgeführt werden, es gibt keine Parallelität. Single-threaded Perfromanz zu verbessern ist schwer, daher sollte die Parallelität verbessert werden.

## Secure Hardware

Sicherstellen von korrekter Datenverarbeitung durch gesicherte Datenverarbeitung im Enclave Page Cache (EPC). Daten sind innerhalb der Enklave verschlüsselt und können nur innerhalb der Enklave entschlüsselt und verarbeitet werden. Ungenutzte Pages werden geswappt, hierfür wird ein Context Switch und ein Integritätscheck gemacht, dafür ist ein großer Overhead notwendig und das OS ist für das Swappen verantwortlich. Intel SGX: