Datenbanksysteme

Abfrageverarbeitung & Query Optimization

Prof. Dr. Patrick Cato

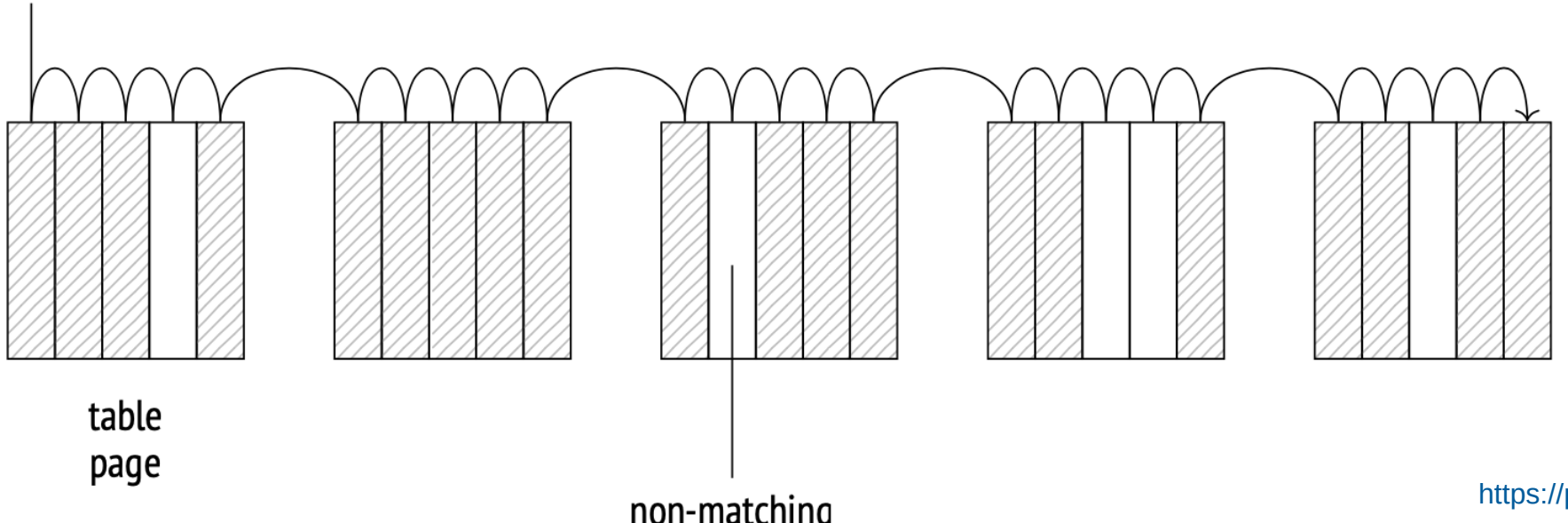
Technische Hochschule Ingolstadt

Theoretische Grundlagen: Begrifflichkeiten

Wiederholung

Sequentieller Scan

Ein sequenzieller Scan ist ein grundlegender Algorithmus zur Suche nach einem bestimmten Element in einer Liste oder einem Array, indem er sequenziell von Anfang bis Ende durchläuft und jedes Element mit dem gesuchten Element vergleicht, bis es gefunden wird oder das Ende der Liste erreicht ist. Im Kontext von PostgreSQL wird das Heap File sequenziell gelesen, ohne dabei einen Index zu verwenden.

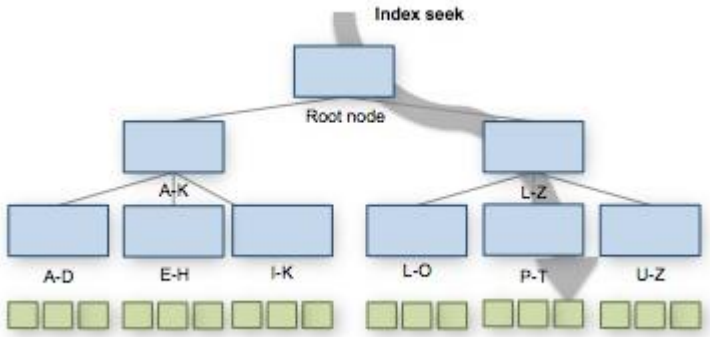


https://postgrespro.com/blog/pgsql/5969403

Wiederholung

Index Scan

In PostgreSQL ist der Index Scan eine Abfrage-Methode, die in der Regel den B-Tree-Index auf einer oder mehreren Spalten einer Tabelle nutzt, um schnell auf die gesuchten Daten zuzugreifen. Der Index Scan arbeitet durch das Durchlaufen des Indexbaums, um den Indexeintrag (oder die Indexeinträge) zu finden, der auf den abgefragten Wert oder den abgefragten Bereich verweist.



Selektivität

Selektivität (sel) im Kontext von Datenbanken ist ein Maß und bezieht sich auf die Fähigkeit einer Abfrage, eine spezifische Menge von Daten aus einer Tabelle zu selektieren, basierend auf bestimmten Filterbedingungen. Es beschreibt das Verhältnis zwischen der Anzahl der Zeilen, die von einer Abfrage zurückgegeben werden, und der Gesamtzahl der Zeilen in der Tabelle.

$$sel_P = \frac{|\sigma_P(A)|}{|A|}$$

Beispiel

ID	Name
1	Meier
2	Smith
3	Miller
4	…
10	Woods

```
select   *   from   mitarbeiter;
```

$$Sel = \frac{10}{10} = 1$$

```
select   *   from   mitarbeiter WHERE id = 1;
```

$$Sel = \frac{1}{10} = 0{,}1$$

Join Algorithmen: Empfehlung auf Online Ressource:

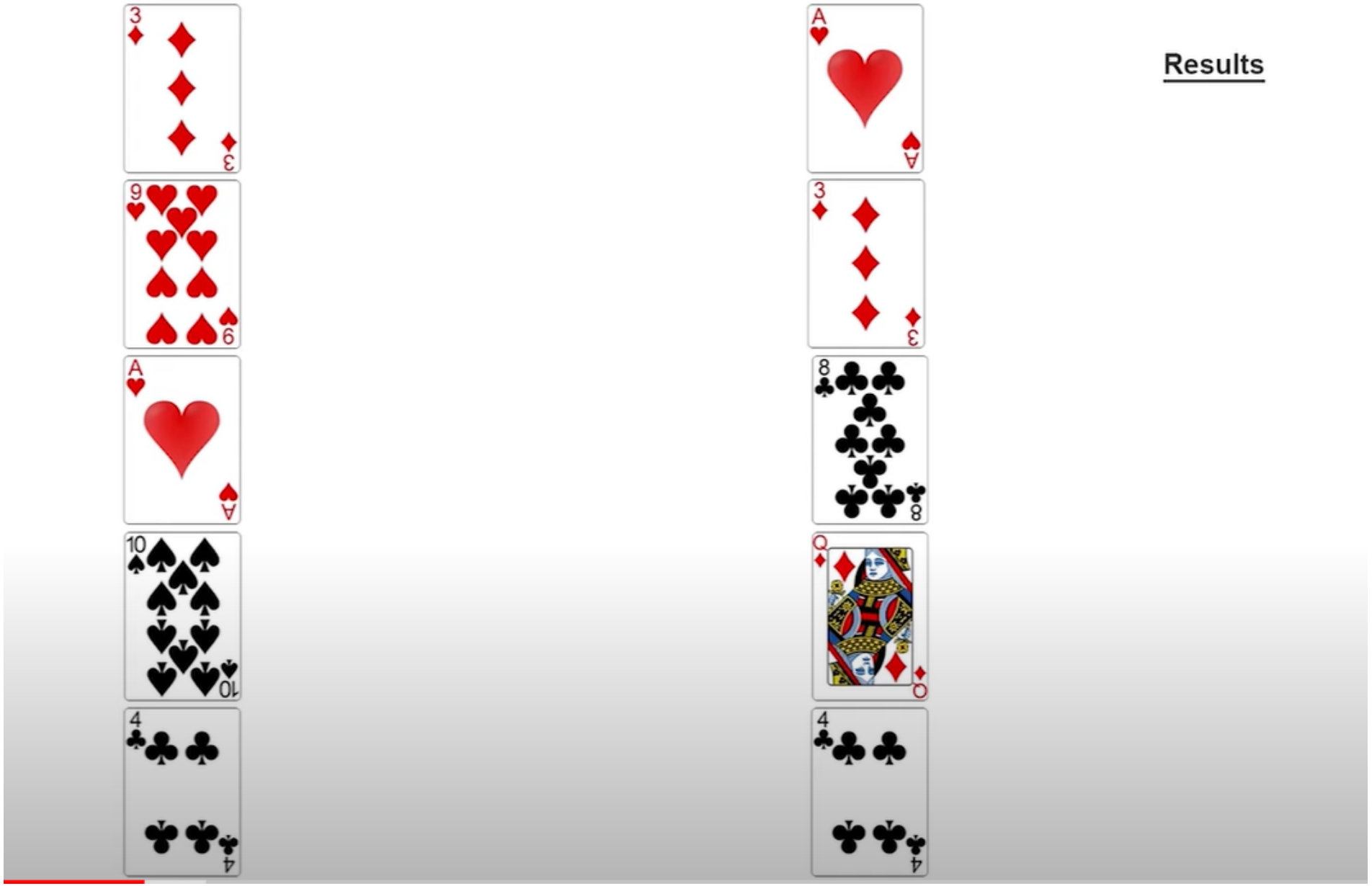
https://www.youtube.com/watch?v=pJWCwfv983Q&t=5s

Join Algorithmen: Nested Loop Join

Nested Loop Join

Ein Nested Loop Join ist eine Methode zur Ausführung einer Join-Operation in Datenbanksystemen, bei der für jeden Datensatz der ersten Tabelle (typischerweise als "Outer" Tabelle bezeichnet) alle Datensätze der zweiten Tabelle (genannt "Inner" Tabelle) durchlaufen und verglichen werden, um passende Paare zu finden, die die Join-Bedingungen erfüllen.

```
for each row R1 in the outer table
for each row R2 in the inner table
if R1 joins with R2
return (R1, R2)
```



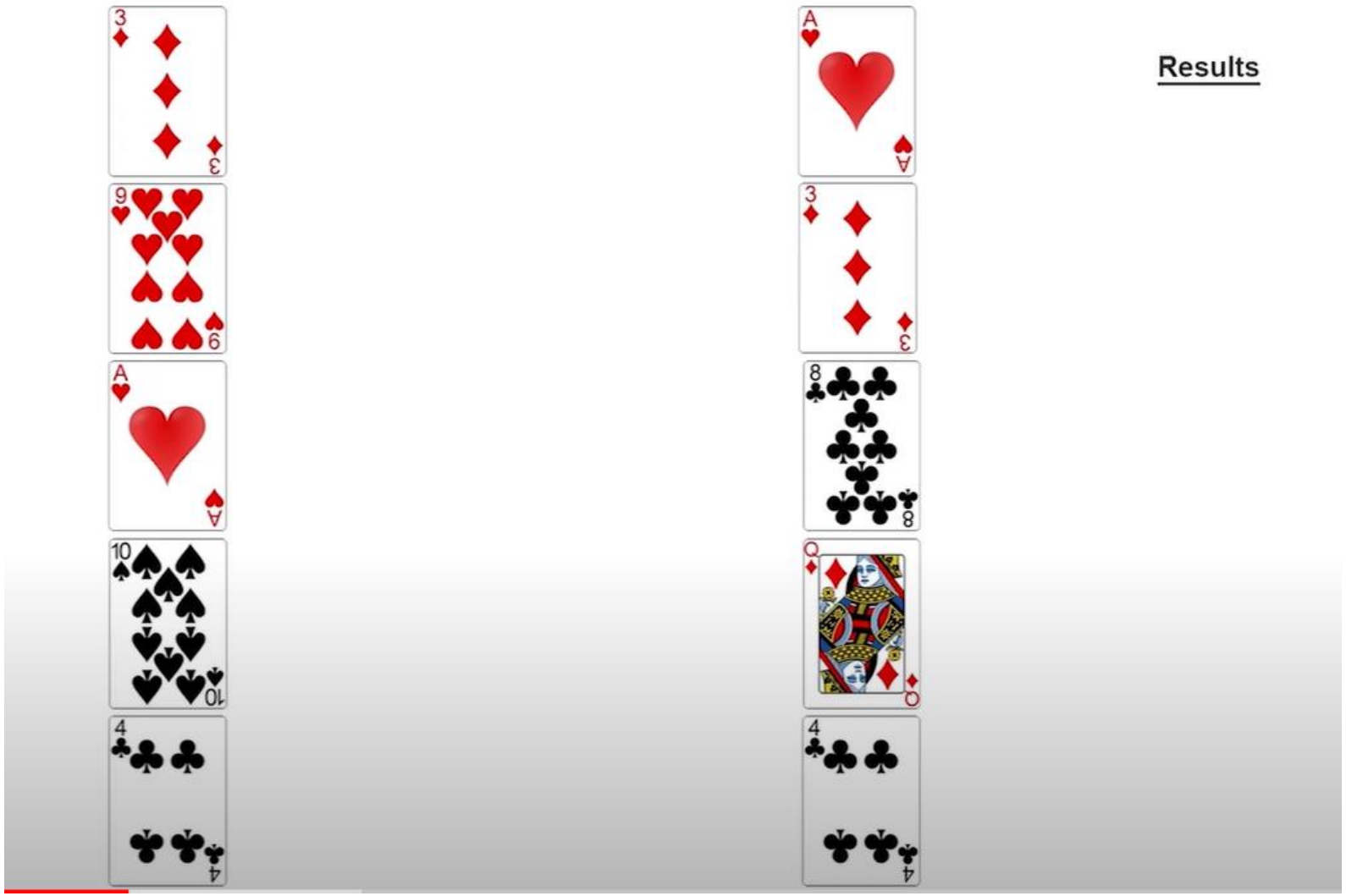
Results

https://www.youtube.com/watch?v=pJWCwfv983Q&t=5s

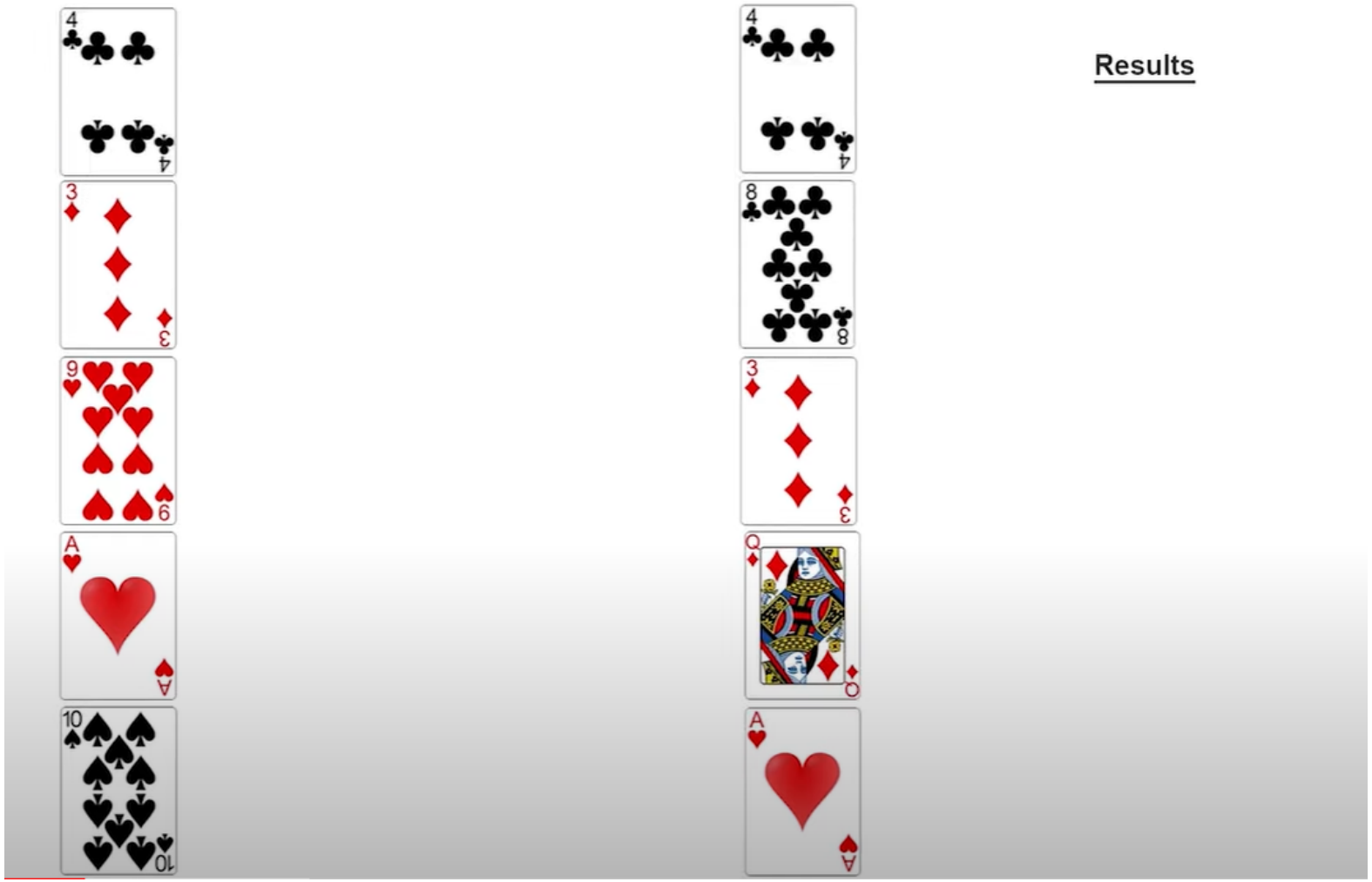
Merge Join (auch Sort Merge Join)

Ein Merge-Join ist ein Datenbank-Join-Algorithmus, der zwei sortierte Eingabedatensätze kombiniert, indem er beide Datensätze parallel sequenziell durchsucht und sie basierend auf einem gemeinsamen Schlüssel fusioniert. Dieser Algorithmus ist effizient, wenn große Datensätze verbunden werden sollen, und er erfordert, dass die Eingabedatensätze vorher nach dem Verbindungsschlüssel sortiert sind.

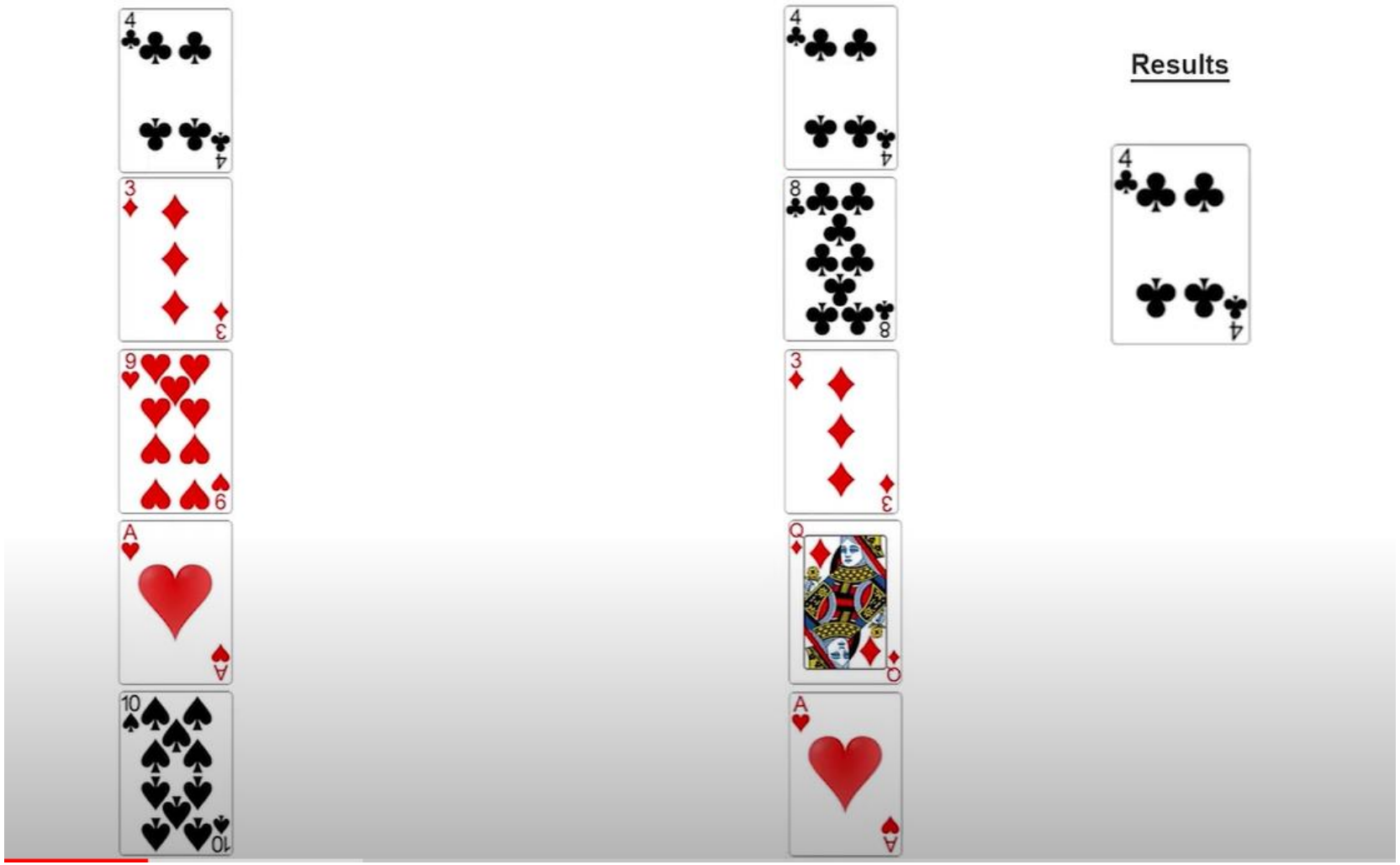
Aussgangssituation

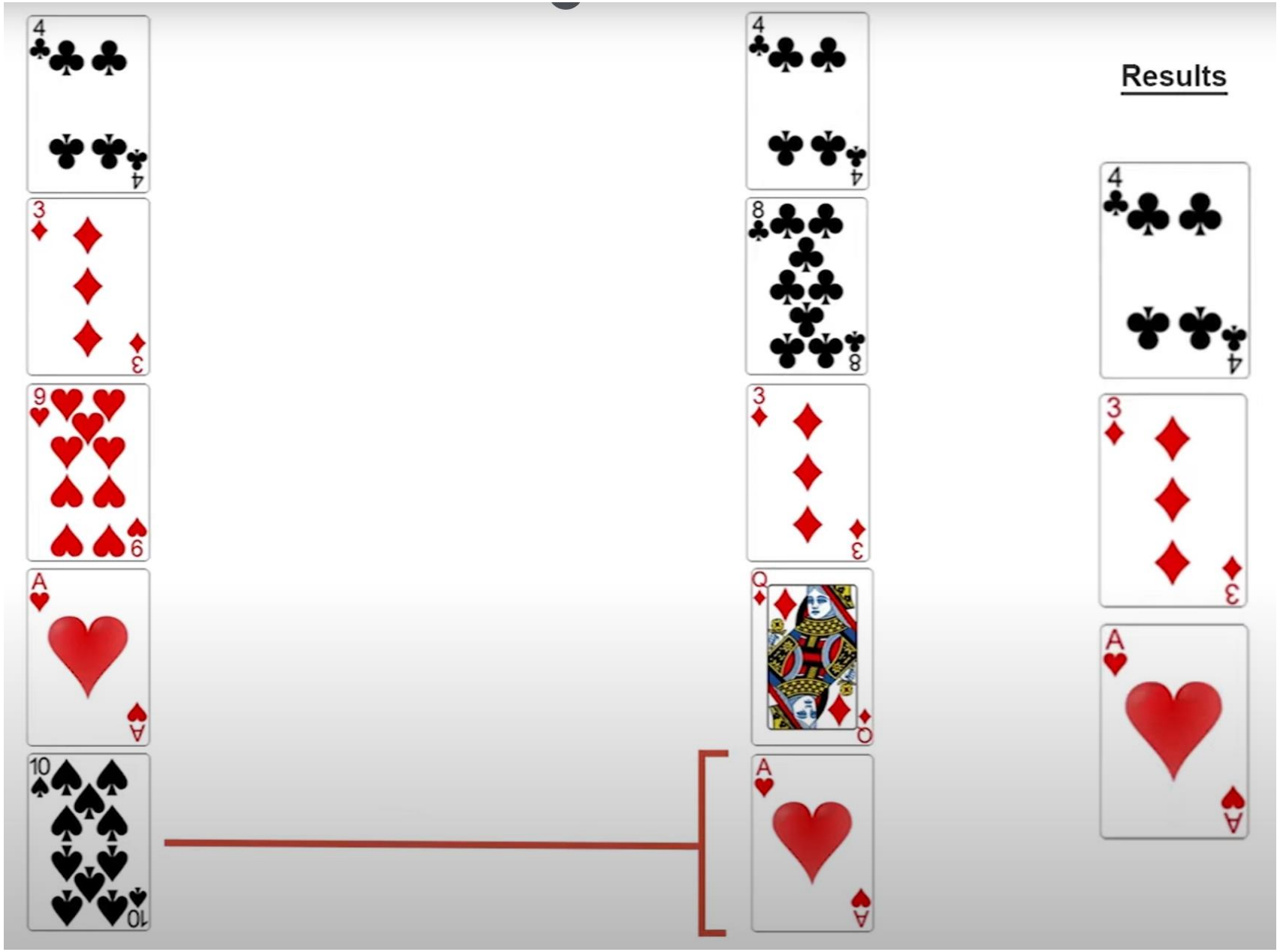


1. Schritt beide Tabellen nach dem Vergleichskriterium sortieren



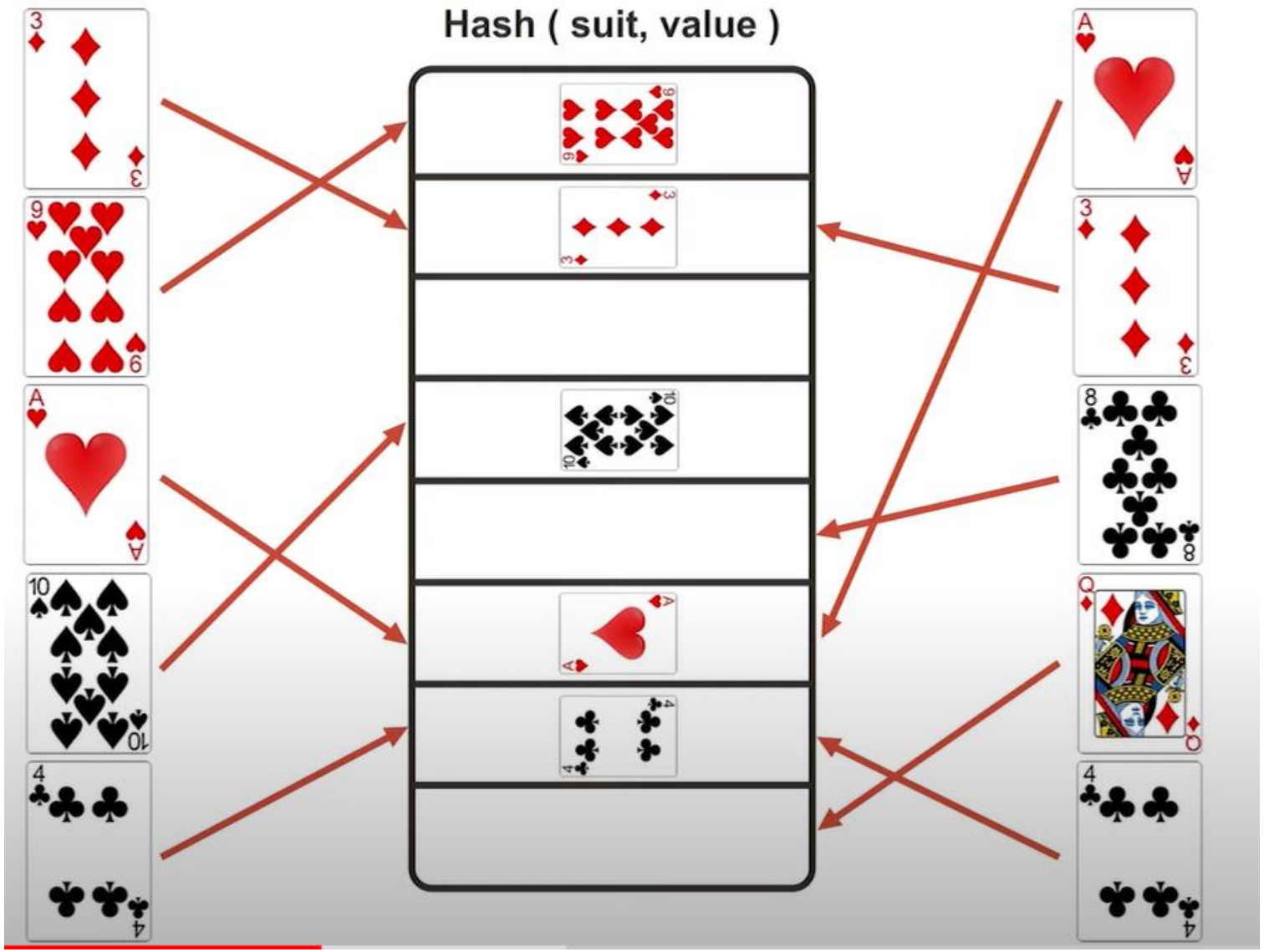
Da Liste sortiert ist können wir stoppen, wenn der Wert von Tabelle2 > Tabelle 1

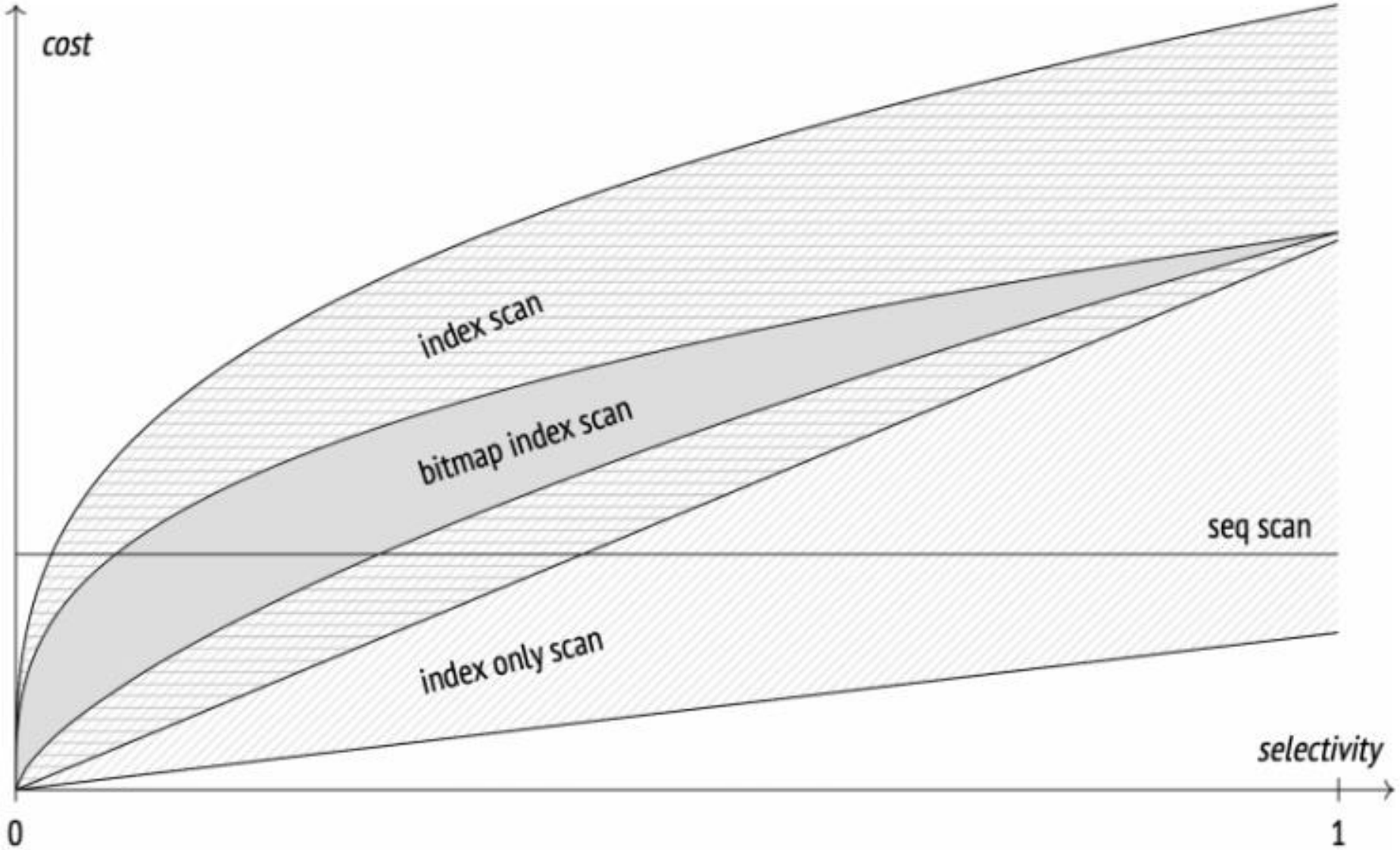



Results

Hash Join

Bei einem Hash Join werden Hash-Tabellen aus den Tupeln einer oder beider beteiligten Relationen erstellt. Anschließend werden diese Tabellen durchsucht, um nur Tupel mit dem gleichen Hash-Code für Gleichheit zu vergleichen. Hash-Joins sind in der Regel effizienter als Nested-Loop-Joins, es sei denn, die Größe der zu durchsuchenden Daten ist sehr klein.


Hash (suit, value)



https://habr.com/ru/companies/postgrespro/articles/666974/

Bitmap Heap Scan & Bitmap Index Scan

Definition

In relationalen Datenbanksystemen wie PostgreSQL treten „Bitmap Heap Scan“ und „Bitmap Index Scan“ in der Regel zusammen auf, insbesondere in **komplexeren Abfragestrukturen**, wo mehrere **Bedingungen und/oder Indizes** involviert sind.

Bitmap / Bitmap Index: Die Bitmap wird dynamisch zur Laufzeit (in-Memory) angelegt und nicht gecached (falls gewünscht BRIN-index anlegen). Eine Bitmap wird auf Basis der Indexe angelegt und gibt an, welche Pages der Tabelle gelesen werden müssen.

Bitmap Heap Scan: Während des Bitmap Heap Scans liest die Datenbank die tatsächlichen Datenseiten aus der Tabelle, die in der Bitmap als relevant markiert wurden.

Index

Key	CTID
Value 1	(2,2)
Value 2	(1,2)
Value 3	(3,2) , (4,1)
Value 4	(2,1) , (3,1)
Value 5	(1,1) , (7,1)
Value 6	(0,1), (0,2), (5,1), (6,1)

Tabelle

CTID	Spalte1	Spalte2
(0,1)	Value 6	Value A
(0,2)	Value 6	Value B
(1,1)	Value 5	Value C
(1,2)	Value 2	Value D
(2,1)	Value 4	Value E
(2,2)	Value 1	Value F
(3,1)	Value 4	Value A
(3,2)	Value 3	Value B
(4,1)	Value 3	Value B
(5,1)	Value 6	Value A
(6,1)	Value 6	Value C
(7,1)	Value 5	Value D

Index

Key	CTID
Value 1	(2,2)
Value 2	(1,2)
Value 3	(3,2) , (4,1)
Value 4	(2,1) , (3,1)
Value 5	(1,1) , (7,1)
Value 6	(0,1), (0,2), (5,1), (6,1)

WHERE Spalte1 = ‘Value6‘

Tabelle

CTID	Spalte1	Spalte2
(0,1)	Value 6	Value A
(0,2)	Value 6	Value B
(1,1)	Value 5	Value C
(1,2)	Value 2	Value D
(2,1)	Value 4	Value E
(2,2)	Value 1	Value F
(3,1)	Value 4	Value A
(3,2)	Value 3	Value B
(4,1)	Value 3	Value B
(5,1)	Value 6	Value A
(6,1)	Value 6	Value C
(7,1)	Value 5	Value D

Bitmap

Page	value
0	1
1	0
2	0
3	0
4	0
5	1
6	1
7	0

Indexe

Key	CTID
Value 1	(2,2)
Value 2	(1,2)
Value 3	(3,2) , (4,1)
Value 4	(2,1) , (3,1)
Value 5	(1,1) , (7,1)
Value 6	(0,1), (0,2), (5,1), (6,1)

Key	CTID
Value A	(0,1), (3,1), (5,1)
Value B	(0,2), (3,2), (4,1)
Value C	(0,3), (6,1)
Value D	(1,1), (7,1)
Value E	(2,1)
Value F	(2,2)

Bitmap

WHERE Spalte1 = ‘Value6‘ OR Spalte2 = ‘ValueA‘

Page	value
0	1
1	0
2	0
3	0
4	0
5	1
6	1
7	0

Page	value
0	1
1	0
2	0
3	1
4	0
5	1
6	0
7	0

Page	value
0	1
1	0
2	0
3	1
4	0
5	1
6	1
7	0

Indexe

Key	CTID
Value 1	(2,2)
Value 2	(1,2)
Value 3	(3,2) , (4,1)
Value 4	(2,1) , (3,1)
Value 5	(1,1) , (7,1)
Value 6	(0,1), (0,2), (5,1), (6,1)

Key	CTID
Value A	(0,1), (3,1), (5,1)
Value B	(0,2), (3,2), (4,1)
Value C	(0,3), (6,1)
Value D	(1,1), (7,1)
Value E	(2,1)
Value F	(2,2)

Bitmap

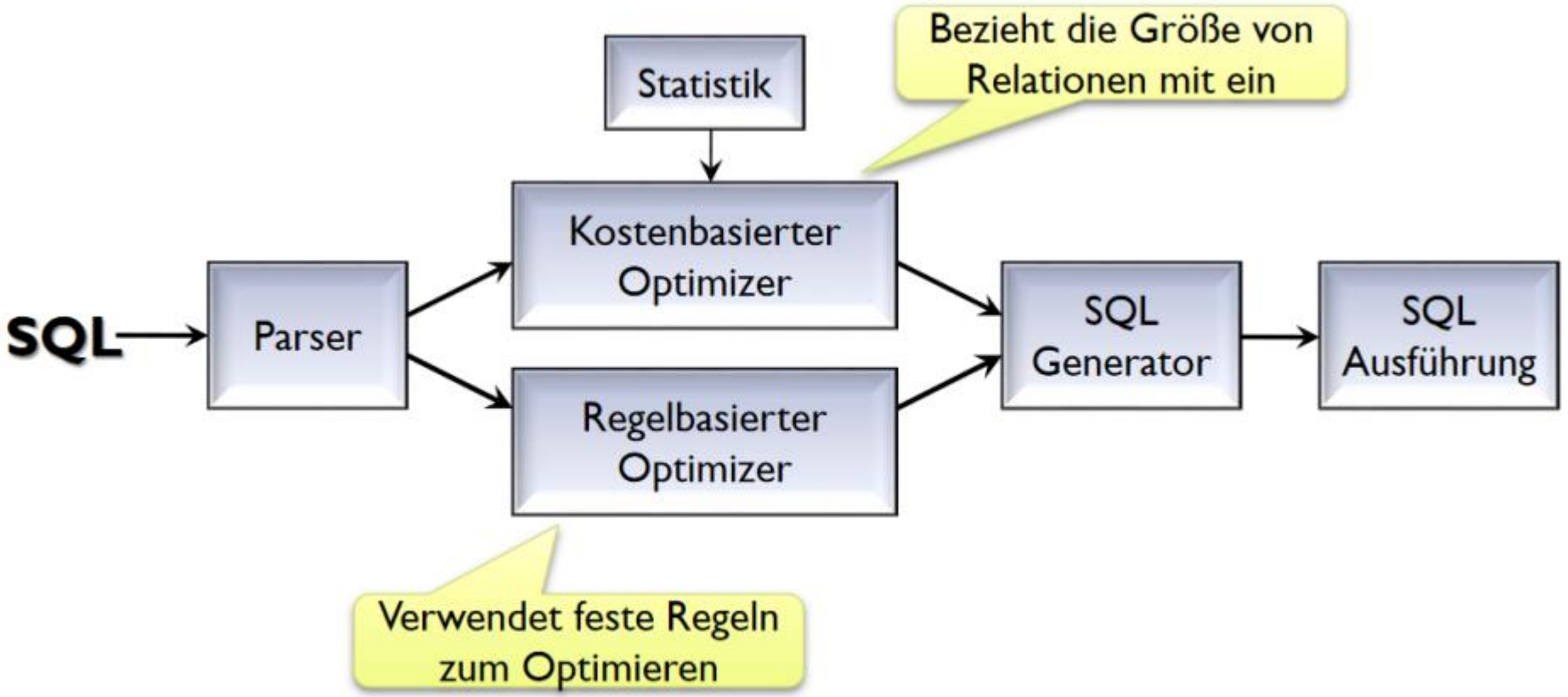
WHERE Spalte1 = ‘Value6‘ AND Spalte2 = ‘ValueA‘

Page	value
0	1
1	0
2	0
3	0
4	0
5	1
6	1
7	0

Page	value
0	1
1	0
2	0
3	1
4	0
5	1
6	0
7	0

Page	value
0	1
1	0
2	0
3	0
4	0
5	1
6	0
7	0

Ausführungsplan



Schicker: Foliensatz-Kap07.pdf

Regelbasierte Optimierung

- **Optimierung nach vorgegebenen Regeln:**

➢ Verwende einen Index, falls vorhanden

➢ Führe erst Projektion durch, dann einen Join

➢ Führe erst Restriktion durch, dann einen Join

➢ Verwende MergeJoin, wenn Relationen bereits sortiert sind

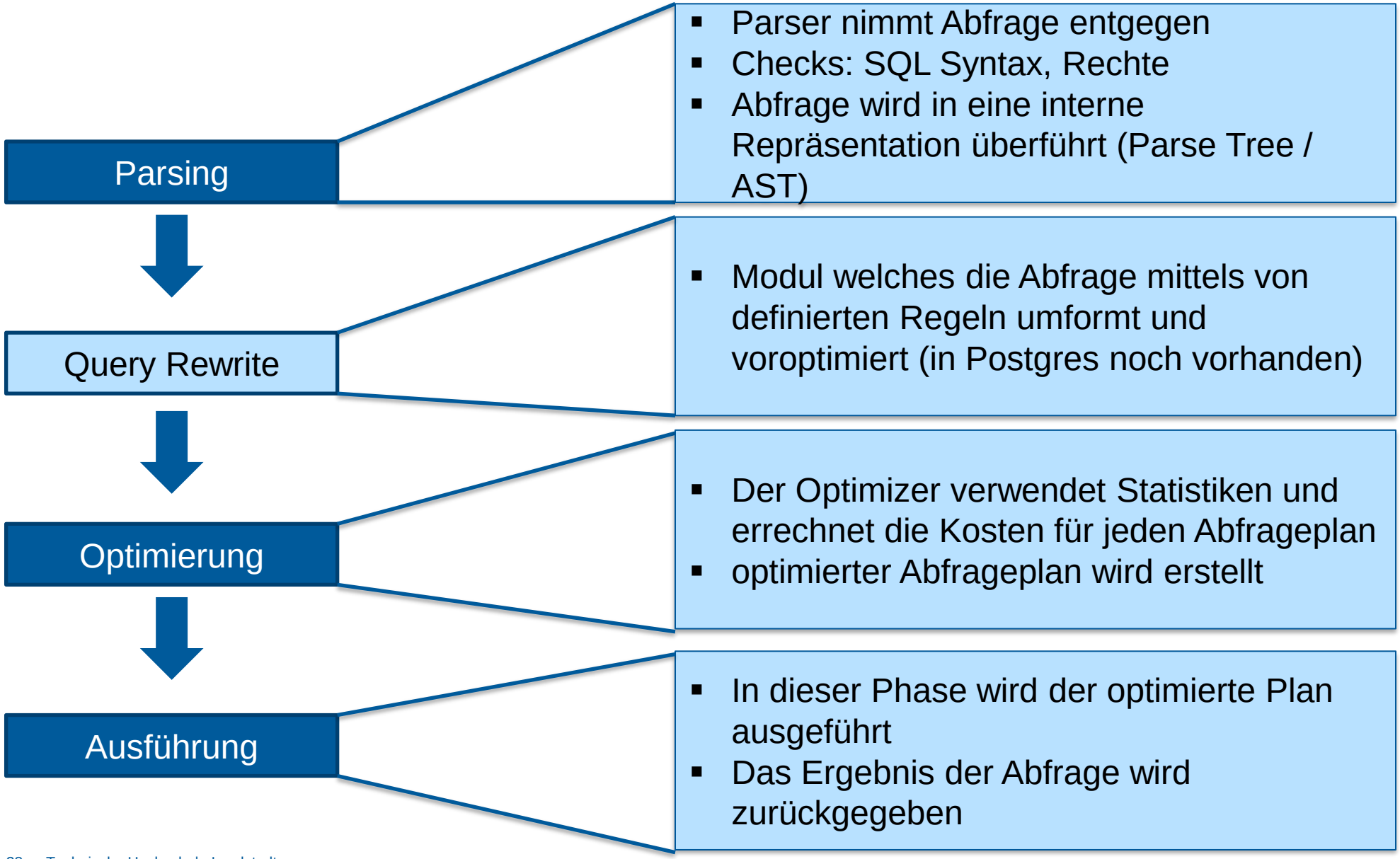
Kostenbasierte Optimierung:

- **Optimierung zusätzlich in Abhängigkeit von den Kosten**

➢ Der Query Optimizer generiert mehrere mögliche Abfrageplan

➢ Query Optimizer verwendet Schätzungen von Kosten, um die Leistung der verschiedenen Abfragepläne zu bewerten und wählt den besten Plan aus

Query Evaluation / Anfragebeverarbeitung



Query Plan

Ein Query Plan ist eine schrittweise Beschreibung, wie eine Abfrage in einer Datenbank ausgeführt wird. Ein Abfrageplan beschreibt, welche Operationen notwendig sind, um die angeforderten Daten aus den verschiedenen Tabellen abzurufen und zu verarbeiten.

Ein Abfrageplan besteht aus einer Abfolge von Operationen wie dem Ausführen von Index Scans, dem Zusammenführen von Daten von mehreren Tabellen, dem Filtern von Datensätzen, dem Sortieren von Ergebnissen usw. Der Abfrageplan zeigt auch die Reihenfolge der Operationen, die verwendet werden, um die Abfrage auszuführen.

Kosten / Costs

Im Kontext von PostgreSQL bezieht sich der Begriff "Kosten" auf eine Schätzung der Ressourcenmenge (Kombination Disk I/O und CPU), die für die Ausführung einer bestimmten Abfrage oder Operation auf einer Datenbank erforderlich ist. Die Kosten werden typischerweise als Maß für die Abfrageleistung verwendet, um verschiedene Ausführungspläne zu vergleichen und den effizientesten Plan auszuwählen

Default Kosten in Postgres für den Optimizer

Parameter	Cost	
cpu_tuple_cost	0,01	Kosten für den Zugriff auf einen Datensatz
cpu_index_tuple_cost	0,005	Schätzung für den Zugriff auf einen Indexeintrag während eines IndexScans
cpu_operator_cost	0,0025	Kostenschätzung für eine Operation oder Funktion einer SQL-Anweisung (z.B. Addition)
seq_page_cost	1	I/O Kosten für das sequenzielle Lesen eines Blocks
random_page_cost	4	I/O Kosten für das Lesen eines einzelnen Datenblocks, zum Beispiel beim Zugriff auf einen Datensatz über einen Index. Normalerweise sind die Kosten für das Lesen eines einzelnen Datenblocks mehr als viermal höher gegenüber dem sequenziellen Lesen.

If the SELECT returns more than approximately 5-10% of all rows in the table, a sequential scan is much faster than an index scan.

Statistiken (Auszug)

Interne Tabelle	Beschreibung
pg_stat_all_ tables	Listet Statistiken für jede Tabelle. Dazu gehören Informationen wie die Anzahl von Full Table Scans sowie die Anzahl von Sätzen, die mittels Table Scan oder Index Scan gelesen wurden.
pg_stat_all_ indexes	Enthält Statistiken über alle Indexe, wie zum Beispiel die Anzahl der ausgeführten Index Scans oder die Anzahl der zurückgelieferten Einträge pro Index Scan
pg_stat_ database	Liefert Statistiken kumuliert auf Datenbankebene. Dazu gehören die Anzahl von Transaktionen sowie die von Datenblöcken, die aus dem Cache und von der Disk gelesen wurden.
pg_stat_database_ conflicts	Statistik über die Anzahl von SQL-Anweisungen, die nicht erfolgreich ausgeführt wurden. Enthält eine Zeile pro Datenbank

In PostgreSQL ist der Default-Wert für die Aktualisierung von Statistiken auf "on" gesetzt. Das bedeutet, dass die Statistiken für eine Tabelle automatisch aktualisiert werden, wenn sich die Daten in der Tabelle ändern. Diese Statistiken werden vom Abfrageoptimierer verwendet, um die effizienteste Ausführungsstrategie für eine Abfrage zu wählen. Wenn die Statistiken veraltet sind, kann dies zu ineffizienten Abfrageplänen führen, die länger dauern oder mehr Ressourcen benötigen, um ausgeführt zu werden.

In einigen DBMS werden Statistiken wöchentlich aktualisiert!

Statistiken

In PostgreSQL werden zahlreiche Statistiken über die Datenbankaktivität und die Charakteristika der gespeicherten Daten erfasst, um die Abfrageleistung zu optimieren. Diese Statistiken werden in verschiedenen Systemkatalogen gespeichert.

1.pg_statistic: Dies ist wahrscheinlich die zentralste Tabelle für die Abfrageoptimierung. Sie enthält Daten über die Verteilung der Werte innerhalb der Spalten von Tabellen, wie etwa Häufigkeiten, Histogramme und Korrelationen zwischen den Spalten. Diese Informationen helfen dem Query Optimizer, die Selektivität von Abfragen besser einzuschätzen.
2.pg_class: Speichert Statistiken auf Tabellenebene, wie die ungefähre Anzahl von Zeilen in der Tabelle und die Größe der Tabelle auf der Festplatte.
3.pg_stats: Bietet eine benutzerfreundlichere Ansicht der Daten in **pg_statistic**, inklusive der meisten dort verfügbaren Statistiken, jedoch in einer einfacher zugänglichen Form.
4.pg_index: Enthält statistische Informationen über die Indizes der Datenbank, wie die Anzahl der eindeutigen Einträge und die Tiefe des B-Baums.
5.Laufzeitstatistiken:

- **pg_stat_user_tables** und **pg_stat_all_tables**: Enthalten Zugriffsstatistiken auf Tabellenebene wie Sequenzscans, Indexscans, Tupel-Inserts, Updates und Deletes.
- **pg_stat_user_indexes** und **pg_stat_all_indexes**: Liefern Statistiken zu Indexnutzung und Effektivität.

```
SELECT * FROM pg_stat_user_indexes;
```

	relid oid	indexrelid oid	schemaname name	relname name	indexrelname name	idx_scan bigint	idx_tup_read bigint	idx_tup_fetch bigint
1	17255	17258	vereinsverwaltung	trainer	trainer_pkey	0	0	0
2	17260	17263	vereinsverwaltung	fussballspieler	fussballspieler_pkey	0	0	0
3	17271	17284	cd	bookings	bookings_pk	0	0	0
4	17274	17286	cd	facilities	facilities_pk	52	52	52
5	17279	17288	cd	members	members_pk	2	2	2
6	17271	17305	cd	bookings	bookings.memid_facid	0	0	0
7	17271	17306	cd	bookings	bookings.facid_memid	112	56091	2511
8	17271	17307	cd	bookings	bookings.facid_starttime	3	0	0
9	17271	17308	cd	bookings	bookings.memid_starttime	1	34	0
10	17271	17309	cd	bookings	bookings.starttime	0	0	0
11	17279	17310	cd	members	members.joindate	4	4	4
12	17279	17311	cd	members	members.recommendedby	0	0	0
13	17644	17647	transaction	accounts	accounts_pkey	0	0	0
14	17650	17654	public	companies	companies_pkey	0	0	0
15	17657	17661	public	contacts	contacts_pkey	0	0	0
16	17767	17773	public	t_big	t_big_pkey	0	0	0
17	17776	17779	public	sales	sales_pkey	0	0	0
18	17783	17786	public	orders	orders_pkey	0	0	0
19	17790	17794	public	sample_data	sample_data_pkey	0	0	0
20	17829	17835	public	t_big3	t_big3_pkey	0	0	0
21	26927	26930	transaktion	accounts	accounts_pkey	18	18	18

```
SELECT * FROM pg_stat_user_tables;
```

	schemaname name	tablename name	attname name	inherited boolean	null_frac real	avg_width integer	n_distinct real	most_common_vals anyarray
32	cd	members	memid	false	0	4	-1	[null]
33	cd	members	surname	false	0	7	-0.8064516	{Smith,Baker,Farrell,Jones}
34	cd	members	firstname	false	0	6	-0.87096775	{David,Darren,Tim}
35	cd	members	address	false	0	28	-0.9677419	{"976 Gnats Close, Reading"}
36	cd	members	zipcode	false	0	4	-0.9354839	{234,4321}
37	cd	members	telephone	false	0	14	-0.9677419	{555-555-5555}
38	cd	bookings	memid	false	0	4	30	{0,3,1,2,8,6,16,5,4,10,21,15,12,7,11,9,14,13,20,17,24,22,29,28,27,30,33,35,26,36}
39	cd	bookings	slots	false	0	4	9	{2,3,1,4,6,9,8}

Query | Query History

```
1  EXPLAIN select * from cd.bookings
2
```

Data Output | Messages | Notifications

	QUERY PLAN text
1	Seq Scan on bookings (cost=0.00..70.44 rows=4044 width=2...

Query | Query History

```
1  EXPLAIN select * from cd.bookings where bookid = 5
2
```

Data Output | Messages | Notifications

	QUERY PLAN text
1	Index Scan using bookings_pk on bookings (cost=0.28..2.50 rows=1 width=...
2	Index Cond: (bookid = 5)

```
1  EXPLAIN select * from cd.members
2
```

ata Output | Messages | Notifications

	QUERY PLAN text
	Seq Scan on members (cost=0.00..1.31 rows=31 width=7...

Query | Query History

```
1  EXPLAIN select * from cd.members where memid =20
2
```

Data Output | Messages | Notifications

	QUERY PLAN text
1	Seq Scan on members (cost=0.00..1.39 rows=1 width=7...
2	Filter: (memid = 20)

```
1183
1184
1185
1186  CREATE TABLE t_big (id serial, name text);
1187
1188  INSERT INTO t_big (name)
1189  SELECT 'Adam' FROM generate_series(1,2000000);
1190
1191  INSERT INTO t_big (name)
1192  SELECT 'Linda' FROM generate_series(1,2000000);
1193
1194
1195  EXPLAIN SELECT * FROM t_big WHERE id = 12345
1196
1197  SHOW
1198
```

Data Output | Explain | Messages | Notifications

	QUERY PLAN text
1	Gather (cost=1000.00..43455.43 rows=1 width=9)
2	Workers Planned: 2
3	-> Parallel Seq Scan on t_big (cost=0.00..42455.33 rows=1 width=9)
4	Filter: (id = 12345)

```
praktikum=# \d t_big
                                Tabelle ╗public.t_big¾
 Spalte |   Typ   | Sortierfolge | NULL erlaubt? |            Vorgabewert
--------+---------+--------------+---------------+-----------------------------------
 id     | integer |              | not null      | nextval('t_big_id_seq'::regclass)
 name   | text    |              |               |
```

Anlage mit Index

```
                              Tabelle ┐public.t_big┐
 Spalte |   Typ   | Sortierfolge | NULL erlaubt? |            Vorgabewert
--------+---------+--------------+---------------+-----------------------------------
 id     | integer |              | not null      | nextval('t_big_id_seq'::regclass)
 name   | text    |              |               |
Indexe:
    "t_big_pkey" PRIMARY KEY, btree (id)
```

```
praktikum=# EXPLAIN select * from t_big WHERE id=5;
                                QUERY PLAN
---------------------------------------------------------------------------
 Index Scan using t_big_pkey on t_big  (cost=0.43..8.45 rows=1 width=9)
   Index Cond: (id = 5)
(2 Zeilen)
```

- **`EXPLAIN ANALYZE`** zeigt den besten Ablaufplan an
- **`EXPLAIN ANALYZE`** führt die Abfrage aus
- **`EXPLAIN ANALYZE`** gibt statistische Informationen aus

Hinweis zu Query Optimization: Secondary Index

- Es kann vorkommen, dass der Optimizer nicht den richtigen Index auswählt wenn die statistischen Daten nicht repräsentativ sind. Dann ist es in SQL möglich den Index gezielt mit anzugeben (wird nicht von Postgres unterstützt):

```
SELECT *
FROM my_table USE INDEX (my_index)
WHERE my_column = 'some_value';
```

- Im Falle von synchronen Indizes werden Schreibzugriffe mit jedem Index langsamer, da erst wenn alle B-Bäume angepasst wurden ein ACK von der Datenbank gegeben wird

Praktische Hinweise

Indexe immer auf Testsystem unter Last prüfen. Nie „probieren“ in Produktion!!! Nicht zu viele Indexe definieren!

- **Suchfelder**: Indexieren Sie Felder, die in WHERE-Klauseln, JOIN-Bedingungen oder in ORDER BY-Klauseln häufig vorkommen. WHERE Clause mit vielen eindeutigen Werten und keine Filterung: Hash Index kann besser sein
- **Spalten mit eindeutigen Werten**: Erstellen Sie Indexe auf Spalten, die eindeutige Werte enthalten, wie z.B. Benutzernamen, E-Mail-Adressen oder Kunden-IDs. Dies kann die Leistung von Abfragen verbessern, die nach einzelnen Datensätzen suchen.
- **Berechnete Felder**: Indexieren Sie berechnete Felder, die in Abfragen verwendet werden, insbesondere wenn komplexe Berechnungen oder Funktionen angewendet werden. Dies kann die Geschwindigkeit von Abfragen verbessern, die auf diesen berechneten Werten basieren.
- **Sortierung:** Erstellen Sie Indexe auf Spalten, die für die Sortierung in ORDER BY-Klauseln verwendet werden, um die Leistung von Sortieroperationen zu verbessern.
- **Indexkombinationen**: In einigen Fällen kann die Erstellung von Indexen auf Kombinationen von Feldern die Abfrageleistung verbessern, insbesondere wenn diese Felder gemeinsam in Abfragen verwendet werden.