Effiziente Matrixmultiplikation

Alexander Weilert, Dragana Jovanovic

Generelles Vorgehen

- ***Vorgehensweise:***
  - Termine und Treffen über Discord
    - Besprechungen
    - Gemeinsame Recherche/Codierungen
    - Absprache ToDos fürs nächstes Treffen
  - Codierung: Try, Error & Retry
  - Recherche: StandUp Meetings
    - Erledigungen
    - Lösungswege

Setting:

Getestet auf:

- Prozessor: AMD Ryzen 7 5800 8-Core Prozessor 3801 MHz
- Grafikkarte: AMD Radeon RX 6800 XT (16GB VRAM)
- Arbeitsspeicher: 32 GB
- Betriebssystem: Windows 10 Pro 64Bit
- Entwicklungsumgebungen:
    - pgAdmin4, Java: 21 (Intellij), Postgresql: 42.7.3

Zeitplanung & Einhaltung

- **Aufwand:**
  - machbar
  - Arbeitsaufwand: 1 Woche vor Phasen Abgabe jeweils 6h/Person
- Für jede Phase alles erfolgreich erledigt
  - Vorarbeit + Nacharbeit

Phase 1

generate()

- generate() Funktion die Tabellen kreiert bzw. löscht, mit angegebener Größe l und sparsity
- generateMatrix: Erstellt Matrizen-Arrays, prüft Sparsity mit Math.random() und fügt zufällige Werte (1-10) ein
- insertMatrix fügt die generierten Matrizen in das DBMS

int [][][] generate (int l, double sparsity)

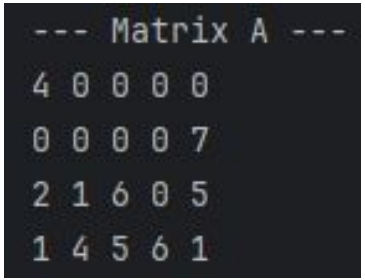
int[][] generateMatrix (int[][] matrix, double sparsity)

insertMatrix (String tableName, int[][] matrix)

– CREATE TABLE A (i INT , j INT , val INT , PRIMARY KEY (i , j))

Tabellen aus generate

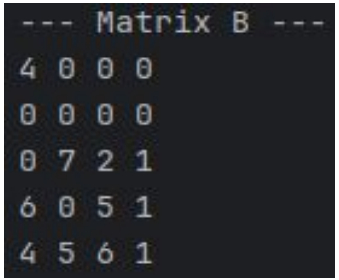
Matrix A als 2D Array



	i	j	val
1	1	1	4
2	2	5	7
3	3	1	2
4	3	2	1
5	3	3	6
6	3	5	5
7	4	1	1
8	4	2	4
9	4	3	5
10	4	4	6
11	4	5	1

Äquivalente Darstellung der Matrizen in der Datenbank ohne Oer

Matrix B als 2D Array



	i	j	val
1	1	1	4
2	3	2	7
3	3	3	2
4	3	4	1
5	4	1	6
6	4	3	5
7	4	4	1
8	5	1	4
9	5	2	5
10	5	3	6
11	5	4	1

Toy-Beispiel - Ansatz 0

- Unser Algorithmus hat eine Laufzeit von $O(n^3)$
  - Algorithmus ist sub-kubisch, wenn Laufzeit kleiner $O(n^3)$, bei uns nicht der Fall
- Funktion beinhaltet drei for-Schleifen, die alle Spalten und Zeilen durchläuft, die jeweils die Länge n besitzen

Funktion: ansatz0 (int[][][] matrix)

```
--- Matrix Calculator ---
16 0 0 0
28 35 42 7
28 67 42 11
44 40 46 12
```

Ansatz 1:

```
public void ansatz1() {  2 usages  Alexander Weilert +1
    try (Statement statement = this.connection.createStatement()) {
        statement.executeQuery( sql: "SELECT a.i, b.j, SUM(A.val * B.val) " +
                "FROM a,  b " +
                "WHERE a.j = b.i " +
                "GROUP BY a.i, b.j");
    } catch (SQLException e) {
        throw new RuntimeException(e);
    }
}
```

- Selection auf A und B mit Berechnung der Multiplikationen

Ansatz 1:

```
EXPLAIN ANALYZE
      SELECT a.i, b.j, SUM(A.val * B.val)
      FROM a, b
      WHERE a.j = b.i GROUP BY a.i, b.j
```

QUERY PLAN

```
1  HashAggregate  (cost=746.59..954.67 rows=20808 width=16) (actual time=0.050..0.077 rows=13 loops=1)
2    Group Key: a.i, b.j
3    Batches: 1  Memory Usage: 793kB
4    ->  Merge Join  (cost=142.69..538.51 rows=20808 width=16) (actual time=0.033..0.039 rows=24 loops=1)
5          Merge Cond: (b.i = a.j)
6          ->  Index Scan using b_pkey on b  (cost=0.15..78.75 rows=2040 width=12) (actual time=0.015..0.016 rows=11 loops=1)
7          ->  Sort  (cost=142.54..147.64 rows=2040 width=12) (actual time=0.015..0.016 rows=23 loops=1)
8                Sort Key: a.j
9                Sort Method: quicksort  Memory: 25kB
10               ->  Seq Scan on a  (cost=0.00..30.40 rows=2040 width=12) (actual time=0.008..0.009 rows=11 loops=1)
11 Planning Time: 0.183 ms
12 Execution Time: 0.385 ms
```

Toy-Beispiel

ansatz0();

	i	j	val
1	1	1	16
2	1	2	0
3	1	3	0
4	1	4	0
5	2	1	28
6	2	2	35
7	2	3	42
8	2	4	7
9	3	1	28
10	3	2	67
11	3	3	42
12	3	4	11
13	4	1	44
14	4	2	40
15	4	3	46
16	4	4	12

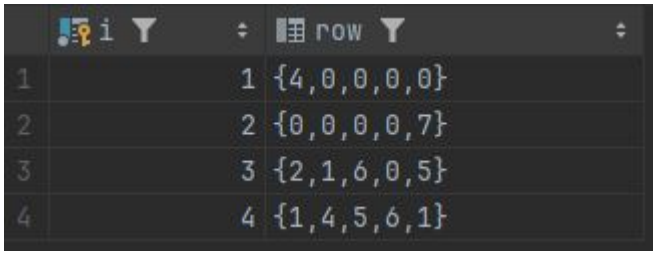
ansatz1();

	i	j	sum
1	1	1	16
2	2	1	28
3	2	2	35
4	2	3	42
5	2	4	7
6	3	1	28
7	3	2	67
8	3	3	42
9	3	4	11
10	4	1	44
11	4	2	40
12	4	3	46
13	4	4	12

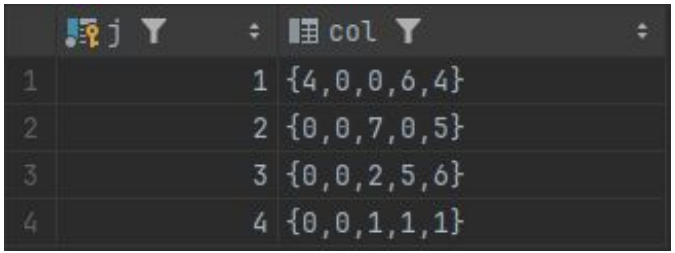
Phase 2

Alternativer Import für Ansatz 2 – einfache Option gewählt

- Funktion createVectorTable, die Tabellen löscht und erstellt
  - Erstellung von zwei Tables “new_A” und “new_B”
- insertAnsatzA befüllt die Zeilenwerte der Matrix in einen Array und gibt diese als “new_A” Table aus
- insertAnsatzB befüllt die Spaltenwerte der Matrix in einen Array und gibt diese als “new_B” Table aus



	i	row
1	1	{4,0,0,0,0}
2	2	{0,0,0,0,7}
3	3	{2,1,6,0,5}
4	4	{1,4,5,6,1}

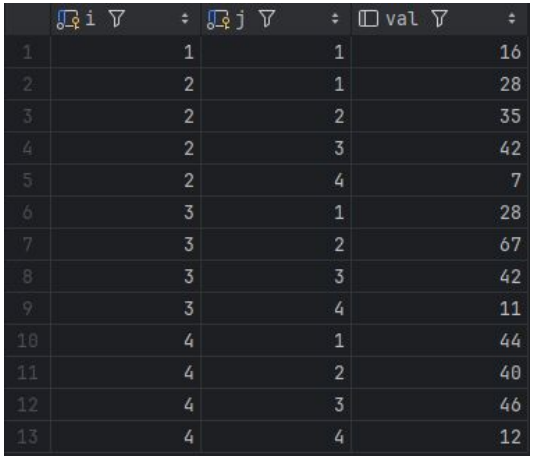


	j	col
1	1	{4,0,0,6,4}
2	2	{0,0,7,0,5}
3	3	{0,0,2,5,6}
4	4	{0,0,1,1,1}

Ansatz 2

- SQL-Funktion dotproduct(), die die Matrixmultiplikation mit Vektoren (new_A, new_B) berechnet
- Kreiert eine Table “new_C” und füllt diese mit dem Ergebnis der Berechnung aus

dotproduct(vector1 int[], vector2 int[])



	i	j	val
1	1	1	16
2	2	1	28
3	2	2	35
4	2	3	42
5	2	4	7
6	3	1	28
7	3	2	67
8	3	3	42
9	3	4	11
10	4	1	44
11	4	2	40
12	4	3	46
13	4	4	12

Ansatz 2: dotproduct()

```
public void createFunction() {  1 usage  Dragana Jovanovic +1
    try (Statement statement = this.connection.createStatement()) {
        statement.execute( sql: "DROP FUNCTION IF EXISTS dotproduct(int[], int[])");
        statement.execute( sql: "CREATE OR REPLACE FUNCTION dotproduct(vector1 int[], vector2 int[]) RETURNS int AS $$\n" +
                "DECLARE\n" +
                "    result int := 0;\n" +
                "BEGIN\n" +
                "    FOR i IN 1..array_length(vector1, 1) LOOP\n" +
                "        result := result + vector1[i] * vector2[i];\n" +
                "    END LOOP;\n" +
                "\n" +
                "    RETURN result;\n" +
                "END;\n" +
                "$$ LANGUAGE plpgsql;");
    } catch (SQLException e) {
        throw new RuntimeException(e);
    }
}
```

Benchmark: Grunddaten

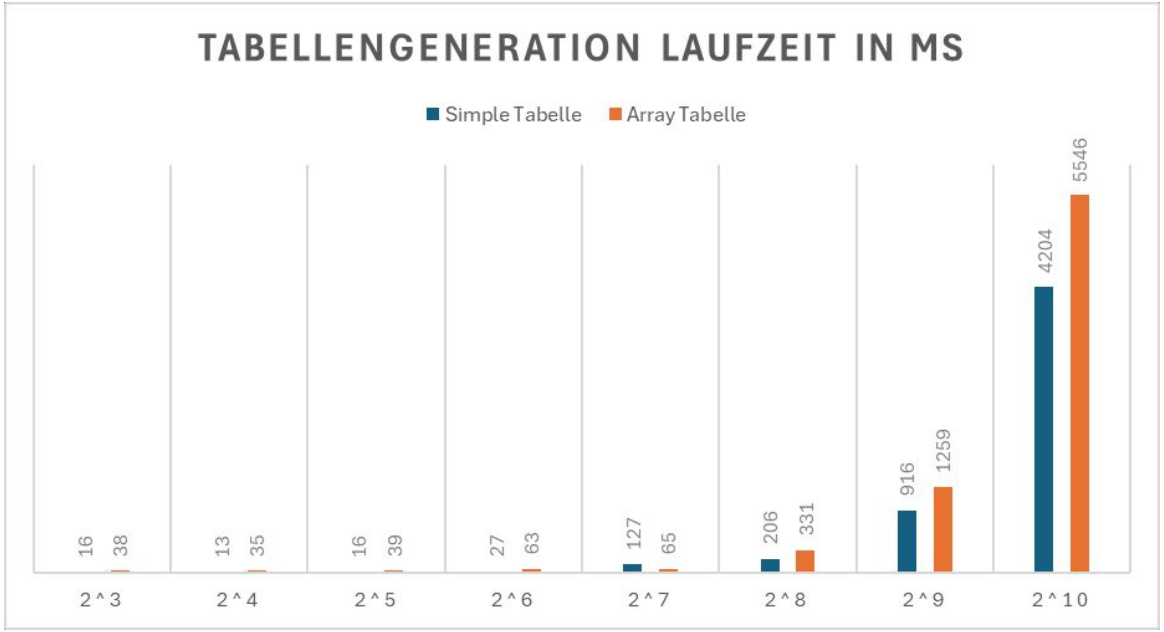
- Benchmark time insgesamt: ~51 min
  - 3 Ansätze á 1 Minute
- L: 8 Testwerte
  - L: 2^3 - 2^10 (inklusive)
  - Sparsity: 0.5 (immer)
- Sparsity: 9 Testwerte
  - L: 2^4
  - Sparsity 0.1 - 0.9 (inklusive)

Testdaten:

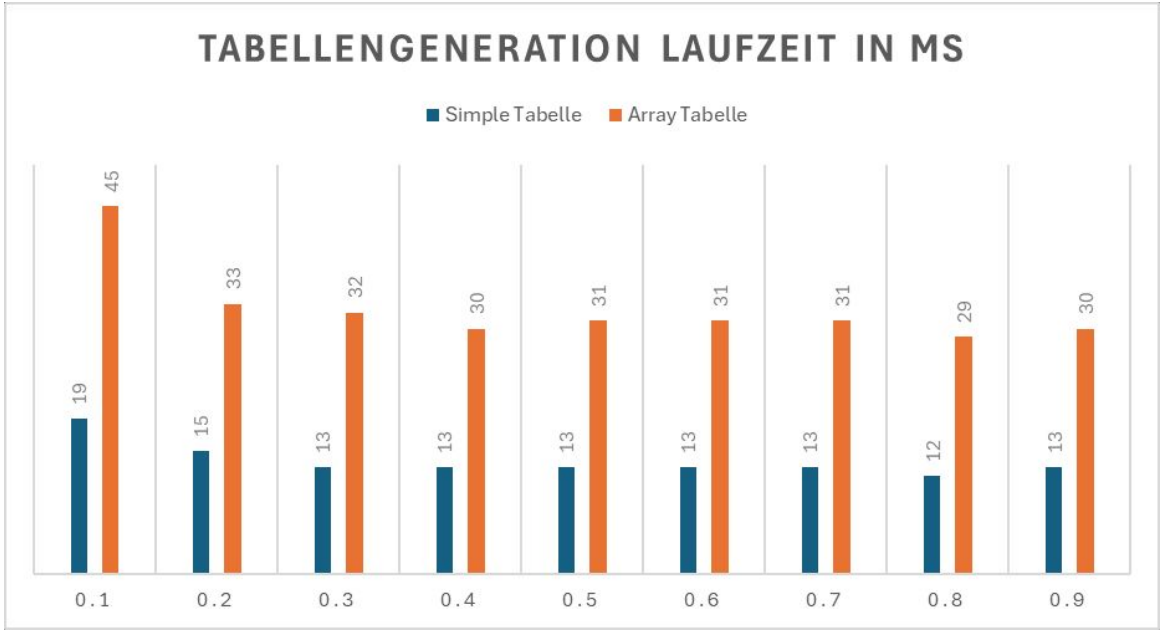
L	Sparsity	Matrix	Array	time in ms	Calc_0	Calc_1	Calc_2
2^3	0.5	16	38	60001	11150	217409	266703
2^4	0.5	13	35	60005	7613	168847	54244
2^5	0.5	16	39	60001	4555	32822	8158
2^6	0.5	27	63	60012	2672	5320	1080
2^7	0.5	127	65	60017	1060	685	140
2^8	0.5	206	331	60381	255	85	19
2^9	0.5	916	1259	60377	57	7	3
2^10	0.5	4204	5546	60063	12	1	0

L	Sparsity	Matrix	Array	time in ms	Calc_0	Calc_1	Calc_2
10	0.1	19	45	60001	7770	86315	53220
10	0.2	15	33	60001	7762	96711	52879
10	0.3	13	32	60001	7858	114706	52930
10	0.4	13	30	60001	7766	144902	53380
10	0.5	13	31	60002	7812	167380	54081
10	0.6	13	31	60002	7797	191930	55799
10	0.7	13	31	60002	7858	238197	60804
10	0.8	12	29	60002	7786	233777	72239
10	0.9	13	30	60005	7822	274409	88414

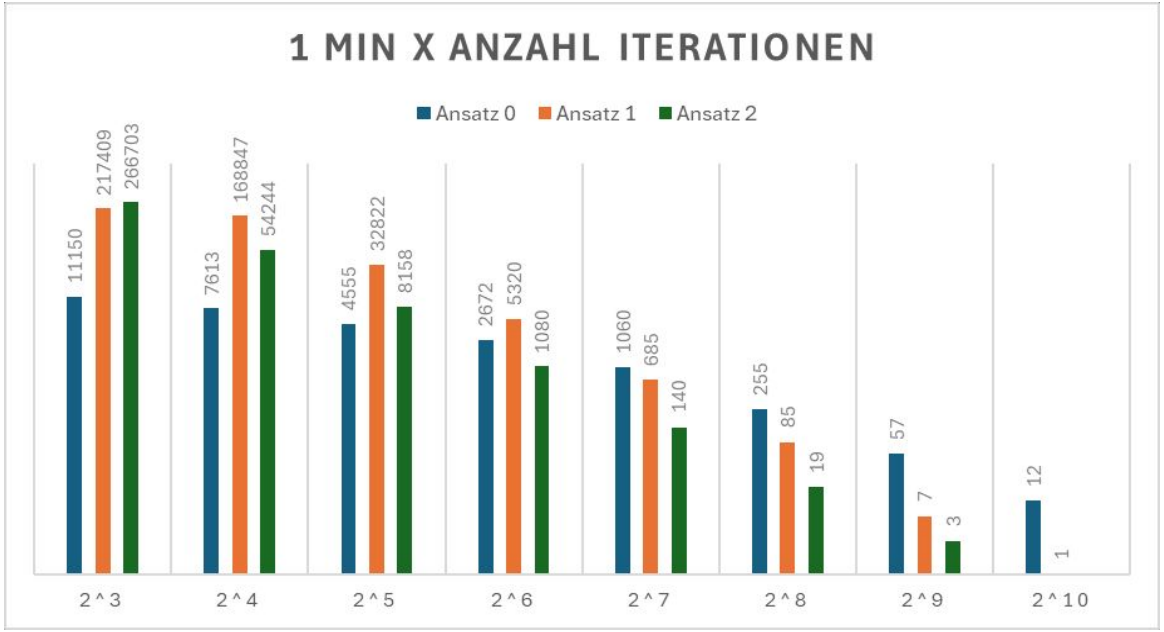
Benchmark - Laufzeit I



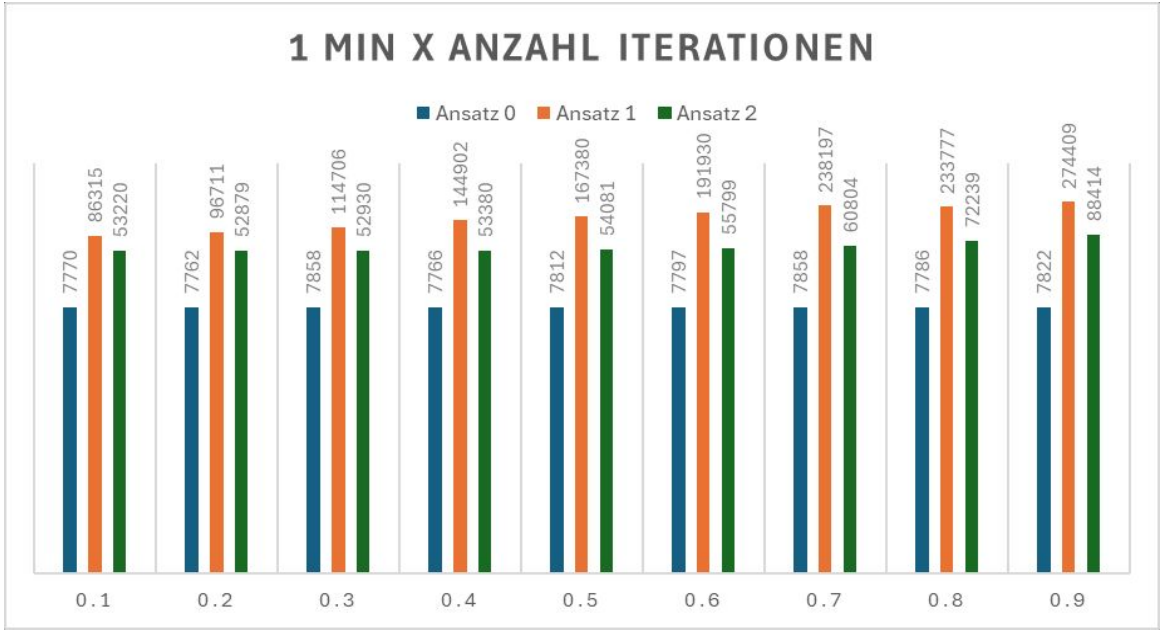
Benchmark - Laufzeit sparsity



Anzahl Iterationen für I



Anzahl Iterationen für sparsity



Was lief gut, was nicht?

- ***Erfolge:***
  - Kommunikation & Teamwork
  - Wiederaufnahme von “alten” Codestücken
  - Implementierung möglichst sauber und organisiert gehalten

Kernergebnis & Fazit

- Auffrischung des Wissens von Linearer Algebra bzw. Matrixmultiplikation
- Danke!

Danke für eure Aufmerksamkeit!