HUMBOLDT-UNIVERSITÄT ZU BERLIN

Scalable Near duplicate search in very large sequence archives

Exposé zur Studienarbeit

Thomas Stoltmann*

10.08.2012

Betreuer: Prof. Dr. Ulf Leser, Astrid Rheinländer

1 Motivation und Zielstellung

Ein zentrales Streben der Molekularbiologie ist es, die Genotypen zu bestimmen, die mit Phänotypen korrelieren [Mar08]. Aus diesem Grund ist die DNA-Sequenzierung seit Jahrzenten ein Forschungsfeld vom besonderen Interesse, das über vielfältige Anwendungen verfügt. Es durchlief einen stetigen Wandel von Einführung der *Didesoxynucleotiden Sequenzierung* nach Sanger et al. [SNC77] an. Die sich in den letzten Jahren entwickelnden sog. *Next Generation Sequencing* (NGS) Techniken stellen derzeit die Spitze dieser Entwicklung dar. Sie verbindet, dass sie im Vergleich zur Sanger-Sequenzierung deutlich kürzere Reads erzeugen, diese jedoch viel schneller zu einem Bruchteil der Kosten produziert werden. So können auf der Roche 454 Plattform 100Mb pro Durchlauf (7h) bei Kosten pro Mb von $84,39 sequenziert werden. Die Länge eines Reads liegt durchschnittlich bei mehr als 250 bp [vgl. Mar08, S. 134]. Nach [Tim+09] liegt der Durchsatz von Geräten GS FLX Roche/454 Serie bei 1,2 GB pro Tag, bei 1 bis 1,5 Millionen verwertbaren Reads pro Durchlauf und einer Leselänge von mehr als 400 Basen. Auf der SOLiD Platform können während eines Durchlaufes (5 Tage) 3000 Mb bei Kosten von $5,81 pro Mb sequenziert werden. Die durchschnittliche Read-Länge beträgt 35 bp [vgl. Mar08, S. 134]. Auf Geräten der SOLiD V3 AB Platform beträgt der Durchsatz 3,2 GB pro Tag, bei 320 Millionen verwertbaren Reads und Leselängen von 25, 35 oder 50 Basen [Tim+09].

Durch die stettige Weiterentwicklung bezüglich der Geschwindigkeit und der Kosten bei der DNA-Sequenzierung bleibt die Weiterverarbeitung der sequenzierten Reads eine Herausforderung. Insbesondere das Alignieren von Reads gegen ein Referenzgenom ist aufwendig und benötigt bei dem weiter steigenden Durchsatz moderner NGS-Geräte immer größere Rechnerkapazitäten.

Tabelle 1 zeigt die Anteile an exakten Duplikaten in einem Datensatz von sequenzierten Exomen kolorektaler Zelllinien. Es ist deutlich zu erkennen, dass der Anteil der exakten Duplikate hoch ist. Eine kurze Untersuchung zur Vorbereitung dieser Studienarbeit auf einem Bruchteil der Daten legt nahe, dass der Anteil der nah-exakten Duplikate vermutlich höher ist. Ein Duplikat gilt als *nah-exakt*, wenn sein Editabstand kleiner gleich 1 ist. Ein Mapping dieser exakten bzw. nah-exakten Duplikate

*stoltman@informatik.hu-berlin.de

bringt für spätere Sequenzanalysen keinen Informationsgewinn, trotz ihres langwierigen Alignments. Im Gegensatz zum Erkennen von exakten Duplikaten, das mit wenig Aufwand effizient erfolgen kann, benötigt das Auffinden von nah-exakten Duplikaten mehr Aufwand. Der in der Voruntersuchung verwendete naive Algorithmus zur Erkennung von nah-exakten Duplikaten erzielte auf bereits 10.000.000 Reads inakzeptable Laufzeiten im Bereich von Monaten. Ziel dieser Arbeit ist es, einen Algorithmus zu entwerfen, der bei großen Mengen von Reads gut skaliert und sich gut parallelisieren lässt. Es gilt dabei zu beachten, dass der Anteil an Cross-Genom-Duplikaten mit steigender Anzahl von Genomen stärker steigen wird. Eine weiteres Ziel dieser Studienarbeit ist es, die Frage zu klären, ob sich Filtern dieser nah-exakten Duplikate überhaupt lohnt, d. h. ob sich das Alignment dadurch beschleunigen lässt.

Gruppe	***Anzahl der Reads***		***Anzahl der Duplikate***	
	mit Duplikaten	***ohne Duplikate***	***absolut***	***relativ***
caco	92849949	83970167	8879782	9,56%
rko	79135892	72816889	6319003	7,99%
lim	88664822	83687770	4977052	5,61%
geo	99143727	92414890	6728837	6,79%
Summiert	359794390	332889716	26904674	7,48%
<Alle zusammen>	359794390	313988627	45805763	12,73%

Tabelle 1: Anteil der exakten Duplikate in einem Datensatz von sequenzierten Exomen kolorektaler Zelllinien. Die Werte der ersten vier Zeilen wurden durch Einzelbetrachtung der Samples caco, rko, lim und geo gewonnen. Diese wurden für die fünfte Zeile jeweils aufsummiert. Die Werte der letzten Zeile wurden gewonnen, indem die Reads aller Samples zusammen betrachtet wurden (u. a. Cross-Genom Duplikate).

2 Verwandte Arbeiten

Im Rahmen dieses Kapitels werden bestehende Arbeiten betrachtet, die die algorithmischen Problemen, die der Studienarbeit zugrundeliegen, behandeln.

2.1 Editabstand und approximatives String-Matching (ASM)

Die Ähnlichkeit bzw. die Verschiedenheit von zwei Reads wird in dieser Studienarbeit mithilfe des Editabstandes beurteilt. Der Editabstand, auch Levenstein-Distanz, ist wie folgt definiert: Der Editabstand zweier Strings s und r ist die minimale Anzahl von Einfüge-, Lösch- und Ersetz-Operationen, um den String s in den String r zu überführen [vgl. Lev66].

Die Algorithmen zur Berechnung des Editabstandes sind eng mit den Algorithmen zur Durchführungn von approximativen String-Matching (ASM) verwandt. Sie teilen sich wesentliche Kernideen und lassen sich nach [Nav01] in folgende Kategorien aufteilen: *Dynamic Programming* Algorithmen [z. B. Sel74; Sel80], Automatentechniken [z. B. Ukk85b; WM92], Filtertechniken und Bit-Parallelisierung [z. B. Mye99].

2.2 Similarity Joins

Similarity-Join Algorithmen lassen sich in zwei grobe Kategorien, die exakten und die appoximativen, einteilen. Letztere lösen das Similarity-Join-Problem approximativ und sind damit für den Zweck der Studienarbeit ungeeignet. Alle hier vorgestellten Algorithmen gehören zur ersten Kategorie, sodass nachfolgend mit Similarity-Join immer ein exakter Similarity-Join bezeichnet wird. Sie lassen sich

folgendermaßen kategorisieren: Zu den Algorithmen, die ein filter-and-refine-Ansatz verwenden, gehören unter anderem Part-Enum [vgl. AGK06], All-Pairs [vgl. BMS07] und ED-Join [XWL08]. Zu den partitionsbasierenden Algorithmen, dessen Kernideen auf einer Partionierung der zu verarbeitenen Strings basieren, zählen Pass-Join [vgl. Li+11] und PartSS [vgl. LSZ11]. Weiterhin existieren noch Algorithmen, die sich einer Baumdatenstruktur bedienen wie PeARL [vgl. RL11] oder Trie-Join [vgl. WLF10; FWL12], die tendenziell eher für kurze Strings gut geeignet sind [vgl. WLF12].

In [Gra+01] wird ein Similarity-Join in bestehenden Datenbankensystemen unter der Verwendung von drei verschiedenen Filtertechniken implementiert. Ziel dieser Filtertechniken ist es, die Anzahl der Kandidatenpaare für die Verifizierung so weit wie möglich zu reduzieren, ohne dabei die Exaktheit des Algorithmus an sich zu ändern. Dazu wird der Editabstand für zwei Strings in schwächere (Filter-) Bedingungen, basierend auf Schnittmenge der beiden q-Gramm-Mengen der jeweiligen Strings, umgewandelt.

[CGK06] implementiert einen Similarity-Join-Operator (*SS-Join*) für Datenbanken mithilfe von q-Grammen, der um einen Präfixfilter erweitert ist. Er ermöglicht die Umgehung des Engpasses, dass ein Kandidatenpaar erst dann verworfen werden kann, nachdem auf alle seine q-Gramme zugegriffen wurde [vgl. XWL08].

Der *PartEnum*-Algorithhmus basiert auf dem Erzeugen von (Hash-)Signaturen durch Partitionierung [vgl. HS11, S. 91ff]. Die Signaturen verfügen über die Eigenschaft, mindestens über eine gemeinsame Signatur zu besitzen, falls für zwei Strings r und s $sim(r, s) \geq \tau$ gilt [vgl. AGK06]. String-Paare mit gemeinsamen Signaturen gelten als Kandidatenpaare und werden in einer späteren Verify-Phase überprüft [vgl. XWL08; AGK06].

All-Pairs ist ein Similarity-Search-Algorithmus, der in Bayardo et al. [BMS07] vorgestellt wird. Er findet ähnliche Vektorpaare mithilfe eines beliebigen Ähnlichkeitsmaßes auf einer Menge von Vektoren eines hochdimensionalen Raums. Er basiert auf einer speziellen Form des *Nested-Loop-Joins*, dem sog. *Index-Nested-Loop-Join*. Er verwendet einen on-the-fly erstellten invertierten Index, der im Hauptspeicher gehalten wird. Mithilfe des invertierten Indexes wird ein Präfix-Filter zur Reduzierung der Kandidatenmenge realisiert.

Eine für den Editabstand modifizierte Version von All-Pairs mit dem Namen *All-Pairs-Ed* wurde von Xiao et al. [XWL08] vorgestellt. Er arbeitet mit q-Grammen, die in dem invertierten Index des Algorithmus gespeichert werden. Zwei Filter reduzieren zusätzlich die Menge der Kandidatenpaare innerhalb der Verify-Phase, bevor ggf. aufwendige Editabstand-Berechnungen durchgeführt werden.

Xiao et al. [XWL08] präsentieren den *Ed-Join* Algorithmus. Sein Grundaufbau basiert auf dem All-Pairs-Ed-Algorithmus. Eine der Kernideen von Ed-Join ist die Analyse der nichtübereinstimmenden q-Gramme anstelle der übereinstimmenden. Aus diesem Ansatz ergeben sich zwei neue untere Schranken, die zu zwei neuen Filtermöglichkeiten führen. In der Evaluation zeigte sich, dass Ed-Join gut für große Schwellwerte von $\tau \in [4, 20]$ und q-Grammgrößen von $q \in [4, 8]$ geeignet ist [vgl. XWL08] und damit für das skizzierte Anwendungsszenario eher ungeeignet ist.

In Xiao et al. [Xia+11] werden zwei Similarity-Join-Algorithmen zur effizienten Erkennung von fast exakten Duplikaten von Dokumenten vorgestellt. Die mit *PPJoin* und *PPJoin+* bezeichneten Algorithmen basieren auf All-Pairs-Ed. Sie verwenden im Gegensatz zu Ed-Join nicht nur q-Gramme, sondern allgemeiner Tokens, die Wörter, q-Gramme usw. darstellen können. Zur Bestimmung der Ähnlichkeit wird die Jaccard-Ähnlichkeit herangezogen.

Zu den partitionierenden Similarity-Join-Algorithmen gehört Pass-Join [vgl. Li+11]. Ihm liegt die Annahme zugrunde, dass sofern ein String s ähnlich zu einem String r ist, es einen Substring von s geben muss, der einem Segment von r gleicht. In der Evaluation von Pass-Join zeigte sich der Algorithmus gegenüber Ed-Join und Trie-Join sowohl für kurze ($l \approx 15$), mittellange ($l \approx 45$) als auch lange Strings ($l \approx 105$) mit $\tau \in \{1, \ldots, 10\}$ überlegen.

Neben Pass-Join gehört auch PartSS aus Li et al. [LSZ11] zu den partitionierenden Algorithmen, dem zusätzlich die Idee der *τ-Neighborhood* nach Myers [Mye94] zugrunde liegt: Als τ-Neighborhood wird die Menge der Strings bezeichnet, die maximal τ-Edit-Operationen von s entfernt sind. Sollte ein String t ähnlich zu einem String s innerhalb eines gegebenen Editabstandes τ sein, so muss dieser in der τ-Neighborhood von s enthalten sein. Da die Größe der τ-Neighborhood $\mathcal{O}\left(|s|^{\tau} \cdot |\Sigma|^{\tau}\right)$ beträgt, ist ihr praktischer Nutzen begrenzt. Bocek et al. [Boc+07] gelang es jedoch einen praktikablen Algo-

rithmus – Fast Similarity Search (*FastSS*) – zu entwickeln. FastSS wird für die approximative Suche in Dictionaries eingesetzt. Er verwendet anstelle der τ-Neighborhood die *τ-Deletion-Neighborhood*, bei der der Editabstand ausschließlich durch Löschungen modelliert wird. Die *τ-Deletion-Neighborhood* eines Strings s enthält dabei alle Varianten von s mit bis zu τ Löschungen. Die Größe der τ-Deletion-Neighborhood beträgt $\mathcal{O}\left(|s|^{\tau}\right)$. Die von FastSS angewendete Filterbedingung für zwei Strings r und s folgt der Idee, dass es mindestens einen gemeinsamen String in den τ-Deletion-Neighborhoods der beiden Strings r und s geben muss, sofern für die zwei Strings s und r $ed(s,r) \leq \tau$ gilt. Für jeden Eintrag des Dictionaries werden die entsprechenden Neighborhoods erzeugt und in einem invertierten Index gespeichert. Das Verfahren ist ausschließlich für kurze Strings und kleine Schwellwerte τ praktikabel [vgl. Wan+09].

Wang et al. [Wan+09] veröffentlichten den *neighborhood generation with partitioning and prefix-based pruning* (*NGPP*) Algorithmus, der den FastSS-Algorithmus von Bocek et al. [Boc+07] erweitert. Im Gegensatz zu diesem zerlegt NGPP jeden String in $\left\lceil \frac{\tau+1}{2} \right\rceil$ Partitionen und speichert die *1-Deletion-Neighorhoods* der Partionen der zu indizierenden Strings in einem invertierten Index. Durch diese Partitionierung wird die Neighborhood-Größe von $\mathcal{O}\left(|s|^{\tau}\right)$ auf $\mathcal{O}\left(|s|\tau + \tau^2\right)$ reduziert. Weiterhin wird durch das neue Präfix-basierende Pruning die Größe der Neighborhood auf $\mathcal{O}\left(l_p \cdot \tau^2\right)$ reduziert, wobei l_p die wählbare Präfixlänge ist. Längen und Positionsunterschiede beim späteren Matchvorgang werden durch Regeln zur Durchführung von Shifting- und Scaling-Operationen ausgeglichen [vgl. Wan+09]. Für kurze Entitäten zeigt sich, dass das Verfahren gegenüber dem klassischen q-Gramm basierenden Verfahren überlegen ist [vgl. Wan+09]. In der Evaluation übertifft NGPP u. a. den FastSS-Algorithmus für kleinere τ sowie Entitäten mit einer Länge von 15 bis 23 Zeichen deutlich [vgl. Wan+09].

Dieses Verfahren wurde schließlich von Li et al. [LSZ11] zu dem *PartSS*-Algorithmus weiterentwickelt, um R-S-Similarity-Joins unter dem Editabstand durchzuführen. Dabei wurde ein neues Partitionsschema (*PlusOne*) eingeführt. Ferner verwendetet PartSS im Gegensatz zu NGPP keine symetrische Vorgehensweise bei der Partitionierung: Zum Indizieren benutzt er beide Partionierungsverfahren, jedoch beim Anfragen lediglich das neu eingeführte PlusOne Schema. In den Evaluationen von Li et al. [vgl. LSZ11] übertrifft PartSS sowohl NGPP als auch aktuelle q-Gramm Ansätze nach [CGK06] und [XWL08].

Allen in diesem Abschnitt dargestellten Algorithmen ist gemein, dass sie nicht für skizzierte Anwendungsszenario dieser Studienarbeit geeignet sind. Sie sind nur für mittlere und größere Schwellwerte ausgelegt oder operieren auf längeren Strings, d.h. Strings mit Längen größer als 30 Zeichen, nicht ausreichend schnell. Jedoch sind die von Wang et al. [Wan+09]; Li et al. [LSZ11] eingeführten Partitionsschemata insbesondere für diese Studienarbeit von Interesse, da mit ihnen womöglich der vorgeschlagene Hash-Algorithmus auch für größere τ angewendet werden kann, ähnlich zu dem Verfahren von Pass-Join [vgl. Li+11].

2.3 Similarity-Joins und das MapReduce-Paradigma

Neben den klassisch Similarity-Join-Algorithmen existieren auch mit MapReduce parallelisierte Algorithmen, die in diesem Abschnitt dargestellt werden.

Der Similarity-Join-Algorithmus von Vernica et al. [VCL10] bzw. Vernica [Ver12] baut auf einem dreistufigen Vorgehen auf und wendet zur Parallelisierung das MapReduce-Paradigma an. Für jede Stufe gibt es genau zwei unterschiedliche Implementierungen, die jeweils aus einer oder zwei MapReduce-Phasen bestehen.

Im ersten Schritt findet eine Sortierung der Token vom Join-Attribut des Records nach ihrer Häufigkeit statt. Die sortierte Tokenliste spielt eine wichtige Rolle für die nachfolgenden Schritte. Die erste Implementierung, das *Basic Token Ordering* (BTO), benötigt zwei Map/Reduce-Phasen. Wohingegen die zweite Implementierung, das *One Phase to Order Tokens* (OPTO), nur einer Phase bedarf, dafür jedoch die Tokenliste komplett im Speicher halten muss.

Im zweiten Schritt werden die Record-ID-Paare (RID Paare) bestimmt, die die gewählte Ähnlichkeitsschwelle überschreiten. Die Ergebnisse werden in Form von Tupeln aus RID des einen Dokumen-

tes sowie des anderen Dokumentes und ihrer Ähnlichkeit zurückgegeben. Beide Implementierungen benötigen eine MapReduce-Phase. Das als *Basic Kernel* (BK) bezeichnete Verfahren verwendetet einen einfachen und selbstetwickelten Similarity-Join-Algorithmus. Das *Indexed Kernel* (PK) Verfahren benutzt für die gleiche Aufgabe den PPJoin+ Algorithmus von Xiao et al. [Xia+11].

Während des dritten Schrittes wird mithilfe der Paarliste aus dem zweiten Schritt, die als ähnlich angesehen werden, das finale Ergebnis aus den Ursprungsdaten extrahiert. Das als *Basic Record Join* (BRJ) bezeichnete Verfahren, verwendet zwei MapReduce-Phasen, um das Ergebnis zu berechnen. Hingegen erledigt das *One-Phase Record Join* (OPRJ) die gleiche Arbeit in einer Phase, indem es die insgesammt 4 Funktionen zu zweien zusammenfasst.

In der Evaluation zeigte sich, dass der beste scale-up für den Self-Join und den R-S-Join die BTO-PK-BRJ-Kombination erreicht wird. Der speed-up war bei allen getesteten Kombinationen gleich [vgl. Ver12, S. 49ff].

In [MF12] wird ein Framework für Similarity-Joins mit MapReduce beschrieben. Es verwendet eine zweiphasige Herangehensweise, die für Sets und Multisets geeignet ist, sich jedoch ggf. auf Vektoren anwenden lässt. Ähnlichkeitsmaße müssen die *Shuffling Invariant Property* erfüllen [vgl. CS02], wie es bei *Nominal Similarity-Messures* (z. B. Jaccard-Similarity) der Fall ist [vgl. MF12].

Weitere MapReduce Umsetzungen von Similiarity-Join-Algorithmen existieren von Elsayed et al. [ELO08] und Baraglia et al. [BML10].

3 Herangehensweise

In der Studienarbeit wird ein exakter Similarity-Join-Algorithmus unter dem Editabstand implementiert, der für sehr kleine Schwellwerte geeignet ist (d. h. $\tau \leq 1$). Er wird auf Hashing basieren.

3.1 Hashing

In Muth et al. [MM96] wurde ein Algorithmus für (exaktes) multiples ASM mittels Hashing vorgestellt. In vielen Evaluationen zeigte sich das Verfahren deutlich gegenüber allen Algorithmen für multiples ASM unter dem Editabstand für $\tau \leq 1$ überlegen. Der Algorithmus entnimmt dabei allen zu suchenden Pattern einen Präfix der Länge p. Für jedes Pattern werden alle möglichen Löschungen erzeugt und gehasht. Beim späteren Matching gleitet ein Fenster mit der entsprechenden Präfixgröße über den Text. Für dieses Fenster werden wiederum alle möglichen Löschungen erzeugt und in der Hashtabelle nachgeschlagen. Sollte ein Treffer erzielt werden, wird eine Verifizierung ausgelöst. Da sowohl die induzierten Pattern als auch die zu anfragenden Pattern jeweils genau eine Löschung enthalten, ist es möglich genau eine Löschung, Insertion oder Substitution abzubilden.

In dieser Studienarbeit soll das Verfahren so modifiziert werden, dass sich daraus eine Hashtabellen-Datenstruktur ergibt, mit der sich effizient Similarity-Joins für $\tau \leq 1$ durchführen lassen. Dazu wird ein Filter-and-verfiy-Ansatz gewählt, bei dem das Hash-Verfahren, die Kandidatenpaare liefert, die später gesondert verifiziert werden.

Sei S eine Menge von Strings mit mindestens der Länge p. Für jedes $s \in S$ wird das p-Präfix $s[1 \dots p]$ extrahiert und die 1-Deletion-Neighborhood (Definition siehe Abschnitt 2.2) dieses Präfixes erzeugt. Für jeden Eintrag aus der Neighborhood wird ein Eintrag in der Hashtabelle angelegt. Es gibt demnach p Einträge für jedes $s \in S$. Die Hashfunktion folgt [MM96] und lautet für einen belieben String x:

$$h(x) = \begin{cases} (h(x[1 \dots |x|-1]) << 1) \oplus r(x[|x|]) & , |x| > 1 \\ r(x[1]) & , |x| = 1 \end{cases} \tag{1}$$

Wobei $r : \Sigma \to \Sigma$ eine beliebige bijektive Funktion ist. Für einen String $x_1x_2x_3x_4x_5$ der Länge 5 ergibt sich folgende Formel zu Berechnung des Hashwertes:

$$h(x_1x_2x_3x_4x_5) = (((r(x_1) << 1 \oplus r(x_2)) << 1 \oplus r(x_3)) << 1 \oplus r(x_4)) << 1 \oplus r(x_5)$$

Das Problem ist, dass zum Berechnen der p Hashwerte insgesamt $\mathcal{O}\left(p^2\right)$ Zeit benötigt wird. Analog zu Muth et al. [MM96] wird ein Trick verwendet, um die Berechnung in linearer Zeit zu erreichen. Der Vorteil dieser Hashfunktion ist, dass nur für einen String aus der jeweiligen 1-Deletion-Neighboorhood der Hashwert vollständig berechnet werden muss — alle weiteren Hashwerte können durch einfache XOR Operationen in $\mathcal{O}(1)$ abgeleitet werden. Folgendes Beispiel verdeutlicht die Berechnung der Hashwerte der 1-Deletion-Neighboorhod für einen String $x_1x_2x_3x_4x_5$:

$$
\begin{aligned}
hash_{x_1x_2x_3x_4} &= h(x_1x_2x_3x_4) \\
hash_{x_1x_2x_3x_5} &= hash_{x_1x_2x_3x_4} \oplus (r[x_4] \oplus r[x_5]) << 0 \\
hash_{x_1x_2x_4x_5} &= hash_{x_1x_2x_3x_5} \oplus (r[x_3] \oplus r[x_4]) << 1 \\
hash_{x_1x_3x_4x_5} &= hash_{x_1x_2x_4x_5} \oplus (r[x_2] \oplus r[x_3]) << 2 \\
hash_{x_2x_3x_4x_5} &= hash_{x_1x_3x_4x_5} \oplus (r[x_1] \oplus r[x_2]) << 3
\end{aligned}
$$

Durch diese speziellen Optimierungen ist es möglich, die Hashwerte der gesamten 1-Deletion-Neighboorhod in $\mathcal{O}(p)$ Zeit zu berechnen [vgl. MM96].

Ähnliche Herangehensweisen, mithilfe von Deletion-Neighborhoods verwendeten schon Bocek et al. [Boc+07] mit FastSS, Wang et al. [Wan+09] mit NGPP o. ä. Für spätere Erweiterungen ist es denkbar, die partitionierende Herangehensweise von NGPP zu adaptieren und so ein Matching für größere τ zu ermöglichen.

3.2 Similarity-Join mittels Hashing

Das Similarity-Join-Problem ist wie folgt definiert:

Problem Similarity-Join: Gegeben seien zwei Mengen R und S von Strings über einem Alphabet Σ, ein Threshold $\tau_{sim} \in [0,1]$ bzw. $\tau_{dist} \in \mathbb{R}^+$ sowie eine Ähnlichkeitsfunktion $\text{sim} : \Sigma^* \times \Sigma^* \to [0,1]$ bzw. eine Distanzfunktion $\text{dist} : \Sigma^* \times \Sigma^* \to \mathbb{R}^+$. Finde alle Paare, für die gilt $(r,s) \in \{(r,s) \in S \times R \mid \text{sim}(r,s) \geq \tau_{sim}\}$ bzw. alle Paare, für die gilt $(r,s) \in \{(r,s) \in S \times R \mid \text{dist}(r,s) \leq \tau_{dist}\}$.

Zur Duchführung des Joins wird R zuerst in der Hashtabelle indiziert. Zum Join mit S wird für jedes Element $s \in S$ die 1-Deletion-Neighborhood des Präfixes $s[1 \ldots p]$ schrittweise erzeugt. Für jedes Element der Neighborhood wird der Hashwert berechnet (vgl. Unterabschnitt 3.1) und in der Hashtabelle nachgeschlagen. Sollte ein Treffer erzielt werden, kann das Paar als Kandidatenpaar angenommen werden. Sollte kein einziges Element der Neighborhood von s einen Treffer erzielen wird s verworfen.

Das Similarity-Self-Join-Problem ist wie folgt definiert:

Problem Similarity-Self-Join: Gegeben seien eine Menge S von Strings über einem Alphabet Σ, ein Threshold $\tau_{sim} \in [0,1]$ bzw. $\tau_{dist} \in \mathbb{R}^+$ sowie eine Ähnlichkeitsfunktion $\text{sim} : \Sigma^* \times \Sigma^* \to [0,1]$ bzw. eine Distanzfunktion $\text{dist} : \Sigma^* \times \Sigma^* \to \mathbb{R}^+$. Finde alle Paare, für die gilt $(s,s') \in \{(s,s') \in S \times S \mid s \neq s' \wedge \text{sim}(s,s') \geq \tau_{sim}\}$ bzw. alle Paare, für die gilt $(s,s') \in \{(s,s') \in S \times S \mid s \neq s' \wedge \text{dist}(s,s') \leq \tau_{dist}\}$.

Seine Durchführung erfolgt, indem S vollständig indiziert wird (vgl. Unterabschnitt 3.1). Anschließend bilden alle Elemente, die sich im gleichen Bucket befinden, jeweils Kandidatenpaare, die später verifiziert werden. Während der Implementierung des Self-Joins soll ggf. überprüft werden, ob es sich lohnt, q-Gramm-Filter auf die Elemente der Buckets anzuwenden, um die Anzahl der Kandidatenpaare weiter zu reduzieren. Ferner müssen optimale Parameter für die Hashtabelle gefunden werden.

3.3 Parallelisierung

Anstelle des klassischen MapReduce-Programmiermodels nach [DG08] soll zur Parallelisierung das PACT-Programmiermodel von [Bat+10] verwendet werden.

Self-Similarity-Join Für die Parallelisierung mithilfe von PACTs wird der Self-Similarity-Join auf eine Map-Reduce-Konstruktion abgebildet. Während der ersten Map-Phase werden die Hashwerte der Neighborhod eines sehr kurzen Präfixes berechnet. Die Ausgabe der Map-Funktion erfolgt als Tupel aus einem Hashwert und einem Read. Die Aufgabe dieses Schrittes besteht primär in der Partionierung der Daten. Die Tupel werden nach Hashwert gruppiert. Diese Art von Partitionierung ermöglicht eine getrennte Bearbeitung in den weiteren Schritten. Während der Reduce-Phase baut jeder Reducer aus seinen Eingabedaten eine vollständige Hashtabelle auf (vgl. 3.1), um die möglichen Kandidatenpaare zu bestimmen und anschließend zu verifizieren. Abbildung 3.3 zeigt dieses Vorgehen. Ein großer Nachteil dieser Lösung ist, dass es möglich ist, dass mehrere Reducer den Editabstand für exakt die gleichen Sequenzpaare berechnen. Um dieses Problem zu umgehen, wäre stattdessen eine Map-Reduce-Reduce-Konstruktion möglich. Dabei bleibt die erste Map-Phase erhalten und die Reduce-Funktion des ersten Vorschlags wird auf eine Map- und eine Reduce-Funktion aufgeteilt, indem die Verifikation in die Reduce-Phase verlagert wird. Abbildung 2(a) zeigt dieses Vorgehen.

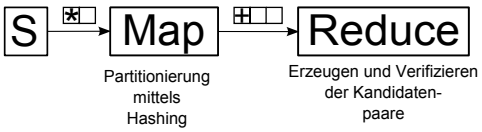
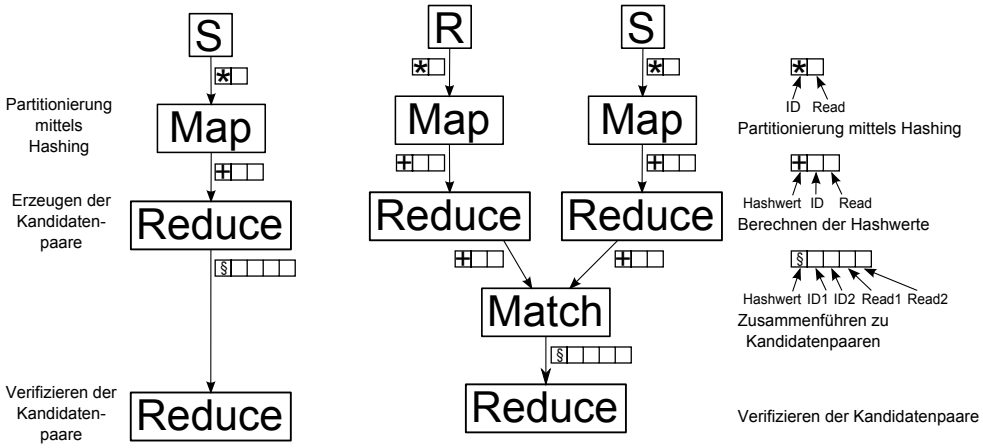


Abbildung 1: Beispielhafter Ablauf für einen Self-Join nach Vorschlag 1.

R-S-Similarity-Join Der R-S-Similarity-Join wird durch eine Map-Reduce-Match-Reduce-Konstruktion implementiert, sodass möglichst viel Funktionalität wiederverwendet werden kann. Dazu durchlaufen die Mengen R und S jeweils separat die beiden Map-Funktionen des Similarity-Joins. Anschließend werden über eine Match-Funktion die Buckets der Hashtabellen zusammengeführt und so die Kandidatenpaare für die anschließende Verifizierung während der folgenden Reduce-Phase erzeugt. Das Vorgehen ist in Abbildung 2(b) verdeutlicht.



(a) Self-Join (b) R-S-Join

Abbildung 2: Darstellung des Self-Joins nach Vorschlag zwei und des R-S-Joins.

3.4 Evaluation

Während der Evaluation sollen für den Self-Similarity-Join und den R-S-Similarity-Join sowohl der Scale-up als der Speed-up untersucht bzw. gemessen werden. Für die Untersuchung der vertikalen

Skalierung wird bei gleichbleibender Datenmenge die Anzahl der Nodes erhöht und die Gesamtlaufzeit betrachtet. Wohingegen bei der Bestimmung des Skalierungsfaktors die Datenmenge und die Anzahl der Nodes um den gleichen Faktor erhöht werden und wiederum jeweils die Gesamtlaufzeit gemessen wird.

Weiterhin wäre es denkbar, gegen einen weiteren Algorithmus zu evaluieren. Gut geeignet wäre hier ein einfacher Algorithmus wie All-Pairs-Ed [XWL08], da sich hier der Implementierungsaufwand in Grenzen halten wird. Ein Evaluieren gegen Algorithmen wie [Ver12; VCL10; BML10; ELO08] ist meines Erachtens wenig sinnvoll, da sie i. d. R. zur Verarbeitung von Dokumenten entworfen wurden und weniger zur Verarbeitung größerer Stringmengen. Ferner würde ihre Implementierung den Zeitraumen der Arbeit überschreiten.

Literatur

Literaturverzeichnis

[AGK06] Arvind Arasu, Venkatesh Ganti und Raghav Kaushik. "Efficient exact set-similarity joins". In: *Proceedings of the 32nd International Conference on Very Large Data Bases, Seoul, Korea, September 12-15, 2006*. VLDB Endowment. 2006, S. 918–929.

[Bat+10] D. Battré, S. Ewen, F. Hueske, O. Kao, V. Markl und D. Warneke. "Nephele/PACTs: a programming model and execution framework for web-scale analytical processing". In: *Proceedings of the 1st ACM symposium on Cloud computing*. ACM. 2010, S. 119–130.

[Bla+10] S. Blanas, J.M. Patel, V. Ercegovac, J. Rao, E.J. Shekita und Y. Tian. "A comparison of join algorithms for log processing in mapreduce". In: *Proceedings of the 2010 international conference on Management of data*. ACM. 2010, S. 975–986.

[BML10] Ranieri Baraglia, Gianmarco De Francisci Morales und Claudio Lucchese. "Document Similarity Self-Join with MapReduce". In: *ICDM 2010, The 10th IEEE International Conference on Data Mining, Sydney, Australia, 14-17 December 2010*. 2010, S. 731–736.

[BMS07] Roberto J. Bayardo, Yiming Ma und Ramakrishnan Srikant. "Scaling up all pairs similarity search". In: *Proceedings of the 16th International Conference on World Wide Web, WWW 2007, Banff, Alberta, Canada, May 8-12, 2007*. 2007, S. 131–140.

[Boc+07] Thomas Bocek, Ela Hunt, Burkhard Stiller und Fabio Hecht. *Fast similarity search in large dictionaries*. Techn. Ber. Department of Informatics, 2007. URL: `http://fastss.csg.uzh.ch/ifi-2007.02.pdf`.

[CGK06] Surajit Chaudhuri, Venkatesh Ganti und Raghav Kaushik. "A primitive operator for similarity joins in data cleaning". In: *Proceedings of the 22nd International Conference on Data Engineering, ICDE 2006, 3-8 April 2006, Atlanta, GA, USA*. 2006, S. 5–5.

[Cha10] Jairam Chandar. "Join Algorithms using Map/Reduce". Magisterarb. University of Edinburgh, 2010. URL: `http://www.inf.ed.ac.uk/publications/thesis/online/IM100859.pdf`.

[CS02] S.H. Cha und S.N. Srihari. "On measuring the distance between histograms". In: *Pattern Recognition* 35.6 (2002), S. 1355–1370.

[DG08] J. Dean und S. Ghemawat. "MapReduce: Simplified data processing on large clusters". In: *Communications of the ACM* 51.1 (2008), S. 107–113.

[ELO08] Tamer Elsayed, Jimmy Lin und Douglas W. Oard. "Pairwise document similarity in large collections with MapReduce". In: *ACL 2008, Proceedings of the 46th Annual Meeting of the Association for Computational Linguistics, June 15-20, 2008, Columbus, Ohio, USA, Short Papers*. The Association for Computer Linguistics. 2008, S. 265–268.

[FWL12] Jianhua Feng, Jiannan Wang und Guoliang Li. "Trie-join: a trie-based method for efficient string similarity joins". In: *VLDB J.* 21.4 (2012), S. 437–461.

[Gra+01] L. Gravano, P.G. Ipeirotis, H.V. Jagadish, N. Koudas, S. Muthukrishnan und D. Srivastava. "Approximate string joins in a database (almost) for free". In: *VLDB 2001, Proceedings of 27th International Conference on Very Large Data Bases, September 11-14, 2001, Roma, Italy*. 2001, S. 491–500.

[HS11] Marios Hadjieleftheriou und Divesh Srivastava. *Approximate String Processing*. Bd. 2. 4. Now Publishers, 2011, S. 267–402. DOI: 10.1561/1900000010.

[Lev66] Vladimir I. Levenshtein. "Binary codes capable of correcting deletions, insertions, and reversals". In: *Soviet physics doklady*. Bd. 10. 8. 1966, S. 707–710.

[Li+11] Guoliang Li, Dong Deng, Jiannan Wang und Jianhua Feng. "Pass-join: A partition-based method for similarity joins". In: *Proceedings of the VLDB Endowment* 5.3 (2011), S. 253–264.

[LSZ11] Zhixu Li, Laurianne Sitbon und Xiaofang Zhou. "PartSS: An Efficient Partition-based Filtering for Edit Distance Constraints". In: *Australasian Database Conference (ADC 2011)*. Bd. 115. 2011, S. 103–112. URL: http://crpit.com/confpapers/CRPITV115Li.pdf.

[Mar08] E.R. Mardis. "The impact of next-generation sequencing technology on genetics". In: *Trends in genetics* 24.3 (2008), S. 133–141.

[MF12] Ahmed Metwally und Christos Faloutsos. "V-SMART-join: a scalable mapreduce framework for all-pair similarity joins of multisets and vectors". In: *Proceedings of the VLDB Endowment* 5.8 (2012), S. 704–715.

[MM96] R. Muth und U. Manber. "Approximate multiple string search". In: *Combinatorial Pattern Matching*. Springer. 1996, S. 75–86.

[Mye86] E.W. Myers. "An O (ND) difference algorithm and its variations". In: *Algorithmica* 1.1 (1986), S. 251–266.

[Mye94] E.W. Myers. "A sublinear algorithm for approximate keyword searching". In: *Algorithmica* 12.4 (1994), S. 345–374.

[Mye99] Gene Myers. "A fast bit-vector algorithm for approximate string matching based on dynamic programming". In: *Journal of the ACM (JACM)* 46.3 (1999), S. 395–415.

[Nav01] G. Navarro. "A guided tour to approximate string matching". In: *ACM computing surveys (CSUR)* 33.1 (2001), S. 31–88.

[RL11] A. Rheinländer und U. Leser. "Scalable Sequence Similarity Search and Join in Main Memory on Multi-Cores". In: *2nd International Workshop on High Performance Bioinformatics and Biomedicine (HiBB), Bordeaux, France* (2011).

[RUN98] M. Ronaghi, M. Uhlén und P. Nyrén. "A sequencing method based on real-time pyrophosphate". In: *Science* 281.5375 (1998), S. 363–365.

[Sel74] P.H. Sellers. "On the theory and computation of evolutionary distances". In: *SIAM Journal on Applied Mathematics* (1974), S. 787–793.

[Sel80] P.H. Sellers. "The theory and computation of evolutionary distances: pattern recognition". In: *J. algorithms* 1.4 (1980), S. 359–373.

[SK04] Sunita Sarawagi und Alok Kirpal. "Efficient set joins on similarity predicates". In: *Proceedings of the ACM SIGMOD International Conference on Management of Data, Paris, France, June 13-18, 2004*. 2004, S. 743–754.

[SNC77] F. Sanger, S. Nicklen und A.R. Coulson. "DNA sequencing with chain-terminating inhibitors". In: *Proceedings of the National Academy of Sciences* 74.12 (1977), S. 5463.

[ST95] Erkki Sutinen und Jorma Tarhio. "On using q-gram locations in approximate string matching". In: *Algorithms—ESA'95* (1995), S. 327–340.

[ST96] Erkki Sutinen und Jorma Tarhio. "Filtration with q-samples in approximate string matching". In: *Combinatorial Pattern Matching, 7th Annual Symposium, CPM 96, Laguna Beach, California, USA, June 10-12, 1996, Proceedings*. Bd. 1075. Springer. 1996, S. 50–63.

[Tim+09] Bernd Timmermann, Marcus W. Albrecht, Vyacheslav Amstislavskiy, Tatiana Borodina, Andreas Dahl, Ralf Herwig, Wilfried Nietfeld, Dmitri Parkhomchuk, Alexej Soldatov, Ralf Sudbrak und Hans Lehrach. "Systematische Analyse der genetischen Variabilität des Menschen". In: *Laborwelt* 3 (2009), S. 8–10.

[Ukk85a] Esko Ukkonen. "Algorithms for approximate string matching". In: *Information and control* 64.1-3 (1985), S. 100–118.

[Ukk85b] Esko Ukkonen. "Finding approximate patterns in strings". In: *Journal of algorithms* 6.1 (1985), S. 132–137.

[VCL10] Rares Vernica, Michael J. Carey und Chen Li. "Efficient parallel set-similarity joins using MapReduce". In: *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2010, Indianapolis, Indiana, USA, June 6-10, 2010*. ACM. 2010, S. 495–506.

[Ver12] Rares Vernica. "Efficient Processing of Set-Similarity Joins on Large Clusters". Diss. UNIVERSITY OF CALIFORNIA, IRVINE, 2012.

[Wan+09] Wei Wang, Chuan Xiao, Xuemin Lin und Chengqi Zhang. "Efficient approximate entity extraction with edit distance constraints". In: *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2009, Providence, Rhode Island, USA, June 29 - July 2, 2009*. 2009, S. 759–770.

[WLF10] Jiannan Wang, Guoliang Li und Jianhua Feng. "Trie-join: Efficient trie-based string similarity joins with edit-distance constraints". In: *Proceedings of the VLDB Endowment* 3.1-2 (2010), S. 1219–1230.

[WLF12] Jiannan Wang, Guoliang Li und Jianhua Feng. "Can we beat the prefix filtering?: an adaptive framework for similarity join and search". In: *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2012, Scottsdale, AZ, USA, May 20-24, 2012*. 2012, S. 85–96.

[WM92] Su Wu und Udi Manber. "Fast text searching: allowing errors". In: *Communications of the ACM* 35.10 (1992), S. 83–91.

[Xia+11] Chuan Xiao, Wei Wang, Xuemin Lin, Jeffrey Xu Yu und Guoren Wang. "Efficient similarity joins for near-duplicate detection". In: *ACM Transactions on Database Systems (TODS)* 36.3 (2011), S. 15.

[XWL08] Chuan Xiao, Wei Wang und Xuemin Lin. "Ed-Join: an efficient algorithm for similarity joins with edit distance constraints". In: *Proceedings of the VLDB Endowment* 1.1 (2008), S. 933–944.