B. Mitschang et al. (Hrsg.): BTW 2017 – Workshopband,
Lecture Notes in Informatics (LNI), Gesellschaft für Informatik, Bonn 2017 365

Serverseitige Aggregation von Zeitreihendaten in verteilten NoSQL-Datenbanken

Oliver Swoboda[1]

Abstract: Die effiziente Erfassung, Abspeicherung und Verarbeitung von Zeitreihendaten spielt in der Zeit von leistungsstarken Anwendungen eine große Rolle. Durch die schnelle und stetig wachsende Erzeugung von Daten ist es nötig, diese in verteilten Systemen abzuspeichern. Dadurch wird es nötig über Alternativen zur sequenziellen Berechnung von Aggregationen, wie Minimum, Maximum, der Standardabweichung oder von Perzentilen nachzudenken. Diese Arbeit untersucht, wie existierende Zeitreihendatenbanken im Hadoop-Ökosystem Aggregationen umsetzen und welche Probleme bei der sequenzuellen Berechnung auftreten. Um diese Probleme zu lösen, wird gezeigt, wie Aggregationen auf Zeitreihendaten verteilt und parallel in verschiedenen Systemen umgesetzt werden können und welche Herangehensweise bessere Laufzeiten liefert.

Keywords: Apache Accumulo, Apache Flink, Zeitreihendaten, Aggregation, serverseitig, verteilt, iterativ

1 Einleitung

Heutzutage werden durch leistungsstarke Anwendungen eine Vielzahl von zeitabhängigen Daten generiert, die es effizient zu erfassen, abzuspeichern und zu Verarbeiten gilt [ADEA11]. Stetig wachsende Datenmengen in Verbindung mit steigenden Anforderungen bei deren Auswertung, lassen konventionelle Herangehensweisen schnell an ihre Grenzen stoßen. Daher werden zunehmend verteilte Lösungen untersucht [ADEA11, Ch10]. Insbesondere ist die parallele und verteilte Berechnung von Aggregationen, wie exakten Perzentilen, eine Herausforderung [SS98].

Die vorliegende Arbeit untersucht einerseits, wie sich die Verwendung unterschiedlicher Schemata in Apache Accumulo[2] auf die Laufzeit von iterativen Aggregatfunktionen auswirken, um herauszufinden ob und in wie weit die Art des Abspeicherns einen Einfluss auf die Performanz der Berechnungen hat. Andererseits wird überprüft, ob eine serverseitige Umsetzung im Datenbanksystem schnellere Berechnungen als in einer Ausführungsumgebung, wie Apache Flink[3], liefert. Außerdem werden Möglichkeiten zur Umsetzung von verteilten und parallelen Aggregationsberechnungen vorgestellt und an einem realen Datensatz evaluiert.

Abschnitt 2 gibt zunächst einen Überblick über verwandte Arbeiten. Abschnitt 3 und Abschnitt 4 stellen das Konzept und die Umsetzung der Aggregationen vor, welche in Abschnitt 5 anhand von realen Daten evaluiert werden.

[1] Universität Leipzig, ScaDS Leipzig, Ritterstrasse 9-13, 04109 Leipzig, mai11brw@studserv.uni-leipzig.de
[2] https://accumulo.apache.org/
[3] https://flink.apache.org/

2 Hintergrund und verwandte Arbeiten

Die sequenzielle Berechnung von Aggregationen über Zeitreihendaten im Umfang von mehreren hundert Gigabyte und mehr, unterliegt einigen Einschränkungen. Heutzutage genutzte Mehrkernprozessoren können nicht optimal genutzt werden, da lediglich ein Thread für Berechnungen ausgenutzt wird. Weiterhin muss genug Festplattenspeicher vorhanden sein, um alle Daten auf einem Rechner ablegen und verarbeiten zu können. Die Größe des Arbeitsspeichers und der maximal mögliche CPU-Takt beeinflussen die Laufzeit der Aggregationen erheblich. Diese Faktoren sorgen dafür, dass sequenzielle Berechnungen mit großen Datenmengen im Vergleich zu parallelen und verteilten Ansätzen teurer sind, da es inzwischen billiger ist mehrere Server mit Standardserverhardware, als einen High-End-Server zu beziehen [Gr09].

Bei der Untersuchung mehrerer Zeitreihendatenbanken im Hadoop-Ökosystem zeigte sich, dass Vertreter, wie OpenTSDB[4] und KairosDB[5], Daten verteilt abspeichern und Aggregationen sequenziell berechnen [Op15, Un15]. OpenTSDB nutzt die NoSQL-Datenbank Apache HBase[6] zum verteilten abspeichern von Zeitreihendaten, wohingegen KairosDB auf Apache Cassandra[7] aufsetzt. Neben Aggregationen, wie Minimum, Maximum, Summe, Anzahl und Durchschnitt, wird auch die Berechnung der Standardabweichung und von Perzentilen unterstützt. Bei KairosDB können diese frei gewählt werden. Wenn mehr als 1028 Werte vorliegen, werden die Perzentile näherungsweise bestimmt [Pe16]. Mit OpenTSDB hat man die Möglichkeit fest vorgegebene Perzentile (50%, 75%, 90%, 95%, 99%, 99.9%) zu berechnen [Op16]. In dieser Arbeit sollen frei wählbare Perzentile, unabhängig von der Anzahl der Werte, exakt berechnet werden. Bei beiden Vertretern werden die verteilten Daten zunächst zu einem Knoten übertragen und anschließend sequenziell berechnet.

Timely[8] ist eine Zeitreihendatenbank der NSA, die Apache Accumulo zum Abspeichern der Daten und zum Berechnen von Aggregationen benutzt. Da Timely erst im Laufe dieser Arbeit veröffentlicht wurde, konnten keine weiteren Untersuchungen vorgenommen werden.

Da Perzentile in dieser Arbeit exakt bestimmt werden sollen, kommen Verfahren, die die Ergebnisse näherungsweise mit einem zufälligen Beispielsatz der Daten berechnen [BS09], nicht in Frage. Um ein Perzentil sequenziell zu bestimmen, müssen die Werte sortiert vorliegen und mit Hilfe der Gesamtanzahl kann dann, z.B. beim Median (50% Perzentil), die gesuchte Stelle $i = \frac{N}{2}$ der Menge A mit $|A| = N$ bestimmt werden, indem über A iteriert wird, bis die gesuchte Stelle erreicht wurde. Im parallelen und verteilten Fall, liegt keine Sortierung der Werte vor und es ist im Allgemeinen nicht möglich, das Ergebnis aus Teilergebnissen zu erhalten. Allerdings gibt es iterative Verfahren für die Berechnung [SS98].

[4] http://opentsdb.net/
[5] https://kairosdb.github.io/
[6] http://hbase.apache.org/
[7] http://cassandra.apache.org/
[8] https://nationalsecurityagency.github.io/timely/

3 Ansatz

3.1 Berechnung in Apache Accumulo

Accumulo bietet ein serverseitiges Programmierframework, die Iteratoren[9]. Dabei handelt es sich im Funktionen, die auf dem Server auf einem oder mehreren Key-Value-Paaren[10] ausgeführt werden. Beim Scanvorgang werden erst die Systemiteratoren, z.B. zum Filtern nach Version oder Zugriffsrechten, und dann Iteratoren vom Nutzer ausgeführt. Ein Iterator erhält die Daten seines Vorgängers. Wenn z.B. ein Iterator die Anzahl der gelesenen Werte liefert, dann kann dessen Nachfolger nicht mehr auf die ursprünglichen Werte zugreifen.

Für die Berechnung von Minimum, Maximum, Summe, Anzahl und Durchschnitt bestimmt jeder Iterator ein Teilergebnis über die gelesenen Daten. Anschließend werden alle Ergebnisse zum Master gesendet und zusammengeführt. Für die Berechnung der Standardabweichung σ müssen normalerweise alle Werte zwischengespeichert werden. Mit der modifizierten Gleichung (1) lässt sich das umgehen, indem die Summen fortlaufend für neue Werte aktualisiert werden. Bei den Perzentilen muss bei der Anwendung des iterativen Algorithmus [SS98] ein Pivotelement, im Allgemeinen der Median der Mediane, bestimmt werden. Hierfür muss jeder Iterator den Median über eine Teilmenge der Daten bilden. Ohne die Daten zwischenzuspeichern, ist das im Allgemeinen nicht möglich, da dazu erst die Anzahl der Werte bestimmt und anschließend erneut über alle Key-Value-Paare iteriert werden müsste. Dafür wäre ein weiterer Scanvorgang notwendig.

$$\sigma = \sqrt{\frac{\sum_{i=1}^{N} A_i^2 - \frac{(\sum_{i=1}^{N} A_i)^2}{N}}{N-1}} \tag{1}$$

Mit Histogrammen ist es möglich, Perzentile in einem Scanvorgang zu berechnen [DZ12]. Hierbei erstellt jeder Iterator ein Histogramm der gelesenen Werte und liefert zusätzlich deren Gesamtanzahl zurück. Anschließend werden die Histogramme zusammengeführt und der gesuchte Rang k, abhängig vom Perzentil, mit der Nearest Rank Methode (Gleichung (2)) bestimmt. Abschließend kann das Ergebnis durch Aufsummieren der Wertanzahlen bestimmt werden. Sobald die Summe der gelesenen Werte größer oder gleich k ist, entspricht die aktuelle Stelle im Histogramm dem gesuchten Perzentil. Die Perzentilberechnung mit Histogrammen ist auf einen kleinen Wertebereich beschränkt. Da es sich bei den genutzten Daten lediglich um ganze Zahlen handelt, die einen Bereich von unter 2000 Werten abdecken, fällt der Speicherbedarf zum Erstellen und Übertragen der Histogramme gering aus. Bei größeren Wertebereichen

$$k = \left\lceil \frac{Perzentil}{100} * Anzahl \right\rceil \tag{2}$$

[9] http://accumulo.apache.org/1.8/accumulo_user_manual.html#_iterators
[10] http://accumulo.apache.org/1.8/accumulo_user_manual.html#_data_model

Metrik	Anzahl Werte
PRCP	907.661.688
TMAX	357.693.332
TMIN	355.857.064
SNOW	317.110.211
SNWD	257.911.061
Andere	389.867.034
Gesamt	2.586.100.390

Tab. 1: Zusammensetzung des Datasatzes

Datenmodell	Row ID	Column Family	Version
Schema1	Monat_SID	Metrik	Tage [1,31]
Schema2	Jahr_SID	Metrik	Tage [1,366]

Tab. 2: Die Schemata

3.2 Datensatz

Bei den Daten handelt es sich um Wetterdaten des Global Historical Climatology Network - Daily (GHCN-Daily) [Me12] und bestehen aus der Stationskennung, dem Datum, der Wettermetrik und dem zugehörigen Wert für den angegebenen Tag. Bei den Wettermetriken werden Niederschlag *PRCP*, Schneefall *SNOW*, Schneetiefe *SNWD*, maximale Temperatur *TMAX* und minimale Temperatur *TMIN* betrachtet, da diese den Großteil der Daten ausmachen, was Tabelle 1 verdeutlicht. Die Daten reichen vom 01.01.1763 bis zum 28.07.2016.

3.3 Datenmodell

Für das Abspeichern der Wetterdaten werden zwei Schemata, die in Tabelle 2 zu sehen sind, genutzt. Tabelle 3 und Tabelle 4 verdeutlichen den Aufbau mit Beispieldaten. *Schema1* nutzt den Monat und die Stationskennung (SID) für die Row ID und legt pro Tag des Monats eine Version an. *Schema2* hingegen nutzt das Jahr in der Row ID und legt pro Tag des Jahres eine Version in der Tabelle an. Beide Schemata speichern die Wettermetriken in der ColumnFamily und nutzen keine Column Qualifiers.

Monat_SID	Metrik	Version	Wert
199401_CA002303986	TMIN	1	-390

Tab. 3: Schema1 am Beispiel

Jahr_SID	Metrik	Version	Wert
1994_CA002303986	TMIN	1	-390

Tab. 4: Schema2 am Beispiel

Durch Locality Groups können Zugriffe auf unterschiedliche Column Families beschleunigt werden. Alle Wettermetriken wurden einer Gruppe zugewiesen, wodurch deren Daten in unterschiedlichen Dateien abgelegt werden. Dadurch müssen die Daten anderer Metriken nicht gelesen und übersprungen werden.

Tabellen werden in Accumulo in Table Splits aufgeteilt. Diese Splits können durch Angabe eines Schwellenwerts eingestellt oder direkt angegeben werden. Pro Split wird maximal ein Iterator ausgeführt. In dieser Arbeit wurde pro Jahr ein Split, also 254 Splits (Zeitraum 1763-2016), angelegt.

4 Implementierung

4.1 Apache Accumulo

In Accumulo wurden die Aggregationen mit einem Iterator in Java umgesetzt. Dieser wird einem BatchScanner[11] hinzugefügt, um die parallele Ausführung zu gewährleisten. Dem Iterator wird vor dem Starten des Scanvorgangs übergeben, welche Aggregation berechnet werden soll. Jede Aggregationsklasse implementiert eine *add*-, *merge*- und *getResult*-Funktion. Während ein Iterator über die Daten iteriert, werden neue Werte durch die Methode *add* dem Aggregator hinzugefügt und dieser letztendlich zurückgeliefert. Anschließend werden die Einzelergebnisse durch die *merge*-Funktion in einem Aggregator zusammengeführt. Das Endergebnis kann dann durch die Methode *getResult* abgerufen werden.

Algorithmus 1 : Pseudocode für die Berechnung des Minimums

```
Data : Wetterdaten verteilt auf x Iteratoren in einem Cluster
Result : Minimum aller Werte
1 min ← ∞
2 forall Iterator iter im Cluster do
3     minIt ← ∞
4     foreach Wert value in iter do
5         if value < minIt then
6             minIt ← value
7     if minIt < min then
8         min ← minIt
```

[11] http://accumulo.apache.org/1.8/accumulo_user_manual.html#_batchscanner

Algorithmus 1 zeigt am Beispiel des Minimums den Ablauf bei der verteilten Berechnung. Hierbei ist zu beachten, dass die Iteratoren parallel ausgeführt werden. Jeder Iterator bestimmt das lokale Minimum *minIt* und liefert das Ergebnis an den Master. Dieser bildet aus den erhaltenen Teilergebnissen das Endergebnis *min* sequenziell. So werden mehrere Milliarden Werte parallel und verteilt voraggregiert, so dass das Ergebnis aus wenigen Hundert Datenpunkten sequenziell bestimmt werden kann. Das Maximum und die Summe werden auf ähnliche Weise berechnet. Beim Durchschnitt wird die Summe und die Anzahl, bei der Standardabweichung zusätzlich noch die Summe der Quadrate der Werte mitgeführt. Beim Abrufen des Ergebnisses wird dann für den Durchschnitt die Summe durch die Anzahl geteilt und bei der Standardabweichung Gleichung (1) angewendet. Für die Berechnung von Perzentilen werden *Maps* mit dem Datenpunkt als Key und einem Zähler als Value genutzt um Histogramme der gelesenen Werte zu erstellen. Auf dem Master werden die *Maps* zusammengeführt um ein Histogramm der Gesamtdaten zu erhalten. Mit Gleichung (2) wird dann der Rang k bestimmt. Anschließend wird über die Werte im Histogramm iteriert, die durch Nutzung einer *TreeMap* sortiert vorliegen. Beim Iterieren werden die Anzahlen addiert und wenn diese gleich oder größer k sind, ist der aktuelle Wert das gesuchte Perzentil.

4.2 Apache Flink

Bei jedem Flinkjob[12] müssen zuerst Daten eingelesen werden. Mit dem *AccumuloInputFormat* können die Daten aus der Datenbank als *DataSet* eingelesen werden. Nach dem Einlesen werden die Daten, die als Key-Value-Paare vorliegen, zunächst mit einer *FlatMapFunction* zu Triplets, welche den Wert, die Anzahl und das Quadrat des Wertes enthalten, transformiert. Für die Verarbeitung erhält jeder Task einen Teil der Daten. Danach werden die erhaltenen Teilergebnisse durch *CombineFunctions* zusammengeführt und das Endergebnis bestimmt.

Minimum, Maximum und Summe sind in Flink bereits implementiert und können auf das erste Element des Triplets angewendet werden. Zur Bestimmung der Anzahl wird die Summe vom zweiten Element berechnet. Der Durchschnitt ergibt sich aus der Summe des ersten Elements und anschließendem summieren der Anzahlen der Werte. Abschließend werden die Teilergebnisse in einer *GroupCombineFunction* aufsummiert und das Ergebnis durch das Teilen der Summe aller Elemente durch deren Anzahl ermittelt. Bei der Standardabweichung werden alle drei Stellen des Triplets benutzt, indem alle enthaltenen Werte jeweils in ihren Tasks aufsummiert und dann wiederum in einer GroupCombineFunction vereint werden. Das Endergebnis wird danach durch Anwenden von Gleichung (1) berechnet.

Bei den Perzentilen wird ähnlich wie in Accumulo vorgegangen. Mit einer *MapPartitionFunction* wird von jedem Task unter Verwendung einer *TreeMap* ein Histogramm der gelesenen Werte erstellt. Diese liefert dann geordnete Paare von *TreeMaps* und Anzahlen der Elemente. Anschließend findet ein *GroupCombine* zum Zusammenführen der *TreeMaps* und Aufsummieren der Anzahlen statt. Außerdem wird an dieser Stelle die Nearest Rank

[12] https://ci.apache.org/projects/flink/flink-docs-release-1.2/concepts/index.html

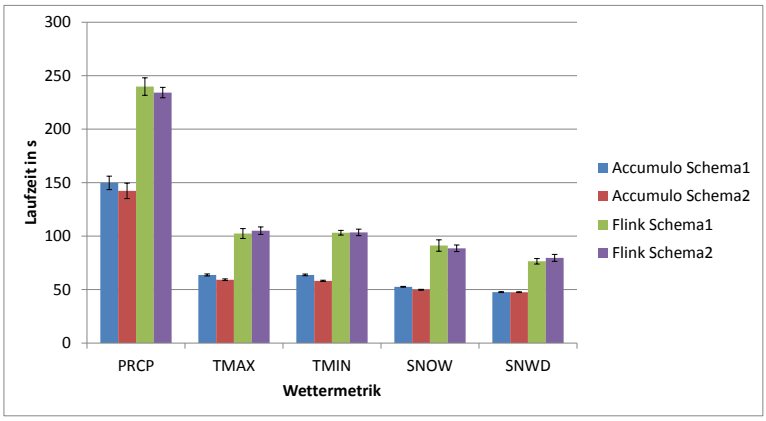

Abb. 1: Ergebnisse der Testläufe

Methode zum Bestimmen des Rangs k angewendet und damit, wie bereits bei Accumulo beschrieben, das Perzentil bestimmt.

5 Evaluation

Um die verschiedenen Schemata und Techniken zu vergleichen, wurde eine Reihe von Tests konfiguriert und ausgeführt. Zunächst wurde überprüft, wie lange ein Scan über die gesamte Tabelle für *Schema1* und *Schema2* jeweils in Accumulo und Flink dauert. Die Ergebnisse hiervon bilden die Baseline. Anschließend wurden die Laufzeiten der implementierten Aggregationen Minimum, Maximum, Summe, Anzahl, Durchschnitt, Standardabweichung und Median, als Vertreter der Perzentile, für die Wettermetriken *TMIN*, *TMAX*, *PRCP*, *SNOW*, *SNWD* gemessen.

Für die Tests stand ein Cluster bestehend aus fünf Knoten zur Verfügung, jeder mit:

- 64 GB Arbeitsspeicher
- 7 TB HDD
- Intel Core i7-6700 (4 Kerne / 8 Threads)
- zwei Netzwerkschnittstellen: extern mit 1 Gbit/s, intern mit 1 Gbit/s
- CentOS Linux release 7.2.1511 (Core)

Dabei fungiert ein Knoten als Master und vier als Worker. Bei der Konfiguration von Accumulo, wurde jedem der vier Tablet Server 16 GB und dem Master 8 GB Arbeitsspeicher zugewiesen. Flink nutzt für jeden der vier Taskmanager 25 GB Arbeitsspeicher.

In Abbildung 1 sind die vorläufigen Ergebnisse der Testläufe zu sehen. Da alle Aggregationen nahezu identische Laufzeiten aufwiesen, wird hier nur die durchschnittliche Laufzeit

des Minimums über den gesamten Zeitraum dargestellt. Für jede Aggregation wurden 10 Testläufe durchgeführt und davon der Durchschnitt und die Standardabweichung berechnet.

Bei beiden Systemen steigt die Laufzeit annähernd proportional zur Datenmenge. Es ist zu sehen, dass Accumulo bei allen Tests deutlich bessere Laufzeiten als Flink erzielt. Ein möglicher Grund dafür ist, dass Flink zuerst die Daten beziehen muss, bevor die Aggregationsberechnungen durchgeführt werden können. Welchen Anteil das Auslesen der Daten mit Flink bei den Testläufen einnimmt, wird Gegenstand zukünftiger Untersuchungen sein. Beim Vergleich der Schemata fällt auf, dass in beiden Systemen kein großer Unterschied zwischen *Schema1* und *Schema2* zu erkennen ist. In Accumulo ist *Schema2* im Durchschnitt schneller. Bei Flink ist jedoch keine klare Tendenz zu sehen.

6 Diskussion

Die ausgeführten Implementierungen in dieser Arbeit sind an die benutzten Daten angepasst. Diese enthalten ausschließlich ganze Zahlen, weswegen bei der Berechnung der Standardabweichung nicht auf Rundungsfehler geachtet werden muss, die bei Fließkommazahlen auftreten würden. Außerdem können dadurch und weil die Werte einen kleinen Wertebereich haben, Perzentile mit Histogrammen der Klassengröße Eins berechnet werden.

Enthalten die Daten Fließkommazahlen, kann es beim Nutzen von Gleichung (1) dazu kommen, dass versucht wird die Wurzel aus einer negativen Zahl zu ziehen[Kn98]. Im allgemeinen Fall muss also eine anderer Algorithmus verwendet werden. Dasselbe gilt für Perzentile, die im Allgemeinen mit einem iterativen Algorithmus berechnet werden müssen, der pro Iteration das Problem verkleinert und gegebenenfalls das Ergebnis schlussendlich sequenziell bestimmt.

7 Fazit und Ausblick

In dieser Arbeit wurde gezeigt, wie iterative Aggregationsberechnungen parallel und verteilt in unterschiedlichen Systemen, Apache Accumulo für die datenbankinterne und Apache Flink für die externe Berechnung, umgesetzt werden können. Es wurde untersucht und evaluiert, wie die implementierten Aggregationen bei steigender Datenmenge skalieren und welches System die besseren Laufzeiten erzielt. Zusätzlich wurde mit zwei Schemata überprüft, ob deren Aufbau einen Einfluss auf die Performanz der Berechnungen hat. Die Ergebnisse zeigen, dass die Laufzeit der Aggregationsberechnungen annähernd proportional zur Datenmenge steigt und Accumulo in allen Testfällen schneller ist. Beim Vergleich der Schemata wurde festgestellt, dass keine großen Unterschiede bei den Laufzeiten festzustellen sind, aber die Berechnungen mit *Schema2* in Accumulo tendenziell schneller ablaufen.

Zukünftig sind weitere Untersuchungen in Bezug auf den zeitlichen Anteil des Einlesens der Daten mit Apache Flink nötig. Weiterhin können Testläufe mit unterschiedlichen Zeiträumen Aufschluss darüber geben, ob sich ein Schema für einen bestimmten Zeitraum besser eignet.

8 Danksagung

Ich möchte mich ganz herzlich bei Matthias Kricke und Martin Grimmer für die Betreuung vor und während meiner Masterarbeit bei mgm technology partners GmbH und an der Universität Leipzig bedanken. Eric Peukert möchte ich für die Betreuung am ScaDS Leipzig und am Lehrstuhl Datenbanksysteme der Universität Leipzig danken. Die vorliegende Arbeit wurde teilweise gefördert durch das Bundesministerium für Bildung und Forschung innerhalb des Competence Center for Scalable Data Services and Solutions (ScaDS) Dresden/Leipzig. (BMBF 01IS14014B)

Literatur

[ADEA11] Agrawal, Divyakant; Das, Sudipto; El Abbadi, Amr: Big data and cloud computing: current state and future opportunities. In: Proceedings of the 14th International Conference on Extending Database Technology. ACM, S. 530–533, 2011.

[BS09] Buragohain, Chiranjeeb; Suri, Subhash: Quantiles on streams. In: Encyclopedia of Database Systems, S. 2235–2240. Springer, 2009.

[Ch10] Chen, Chun; Chen, Gang; Jiang, Dawei; Ooi, Beng Chin; Vo, Hoang Tam; Wu, Sai; Xu, Quanqing: Providing scalable database services on the cloud. In: International Conference on Web Information Systems Engineering. Springer, S. 1–19, 2010.

[DZ12] Dinkel, Kevin; Zizzi, Andrew: Fast Median Finding on Digital Images. In: AIAA Regional Student Paper Conference. April. Jgg. 4, 2012.

[Gr09] Greenhalgh, Adam; Huici, Felipe; Hoerdt, Mickael; Papadimitriou, Panagiotis; Handley, Mark; Mathy, Laurent: Flow processing and the rise of commodity network hardware. ACM SIGCOMM Computer Communication Review, 39(2):20–26, 2009.

[Kn98] Knuth, Donald Ervin: The Art of computer programming. Volume 2, Seminumerical algorithms. S. 216, 1998.

[Me12] Menne; M.J.; Durre, I.; Korzeniewski, B.; McNeal, S.; Thomas, K.; Yin, X.; Anthony, S.; Ray, R.; Vose, R.S.; Gleason, B.E.; Houston, T.G.: Global historical climatology network-daily (GHCN-Daily), Version 3.22. NOAA National Climatic Data Center, 2012. http://doi.org/10.7289/V5D21VHZ, Stand:18.10.2016.

[Op15] OpenTSDB Aggregators.java, https://github.com/OpenTSDB/opentsdb/blob/e2efe9a4875f8cddd3404e9c6b64f5aab4b72c0a/src/core/Aggregators.java#L184, Stand: 18.10.2016.

[Op16] OpenTSDB Aggregators, http://opentsdb.net/docs/build/html/user_guide/query/aggregators.html, Stand: 18.10.2016.

[Pe16] Percentile aggregator returns inconsistent values for longer sample lengths, https://github.com/kairosdb/kairosdb/issues/290,Stand:18.10.2016.

[SS98] Saukas, Einar LG; Song, Siang W: Efficient selection algorithms on distributed memory computers. In: Proceedings of the 1998 ACM/IEEE conference on Supercomputing. IEEE Computer Society, S. 1–26, 1998.

[Un15] Understanding KairosDB for production (distributed database)?, https://groups.google.com/d/msg/kairosdb-group/Pb2W9CXsSkY/0NFHOyKKCgAJ, Stand: 18.10.2016.