UNIVERSITÄT
KOBLENZ · LANDAU
Fachbereich 4: Informatik

Dynamische Aktualitätsbewertung von statistischen Informationen bei der Optimierung föderierter SPARQL-Queries auf veränderlichen Datenquellen

Diplomarbeit

zur Erlangung des Grades eines Diplom-Informatikers
im Studiengang Informatik

vorgelegt von

Dominik Brosius

Matrikel-Nr. 205210161

Erstgutachter:	Prof. Dr. Steffen Staab Institut für Web Science and Technologies
Zweitgutachter:	Dipl.-Inform. Olaf Görlitz Institut für Web Science and Technologies

Koblenz, im September 2011

Erklärung

Ich versichere, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe.

	Ja	Nein
Mit der Einstellung der Arbeit in die Bibliothek bin ich einverstanden.	☐	☐
Der Veröffentlichung dieser Arbeit im Internet stimme ich zu.	☐	☐

.....

(Ort, Datum) (Unterschrift)

Abstract

A trending topic in Semantic Web research deals with the processing of queries over Linked Open Data (LOD). As has been shown in literature, the loose nature of the „web of data“ and data sources within can be accounted for by employing federated query processing strategies. This approach, however, is all the more dependent on both up-to-date statistical summaries (data statistics) of the sources in use and accurate and precise estimation of cardinalities and selectivities. In general, federated data sources are to be seen as black-boxes w.r.t. data statistics, as no interchange of such information can be expected. Because of this, it is possible for individual data statistics to become obsolete, if the corresponding source is subjected to data changes cumulating over time. In this thesis an adaptive system is being proposed, that complements a given RDF-based query federator. Through observation and analysis of the error of the cardinality estimation of incoming queries, it tries to infer the obsolescence of individual data statistics, triggering updates of data statistics found to be obsolete. An evaluation of the system shows, that the approach to this solution is plausible. Yet, in practice no satisfying results could be acquired, that would prove a true practicality. Still, parts of the system proposed may be re-used for related tasks that could be more promising.

Zusammenfassung

Ein neueres Thema innerhalb des Forschungsbereichs Semantic Web behandelt die Verarbeitung von Anfragen über Linked Open Data (LOD). Wie in der Literatur bereits diskutiert wurde, lässt sich der losen Zergliederung innerhalb des „Web of Data“ und dessen Datenquellen durch moderne föderierte Verarbeitungsstrategien bezüglich eingehender Anfragen begegnen. Dieser Ansatz ist jedoch umso mehr abhängig von aktuellen statistischen Informationen (Datenstatistiken) über sämtliche der benutzten Datenquellen einerseits, und genauen Schätzungen von Kardinalitäten und Selektivitäten andererseits. Da föderierte Datenquellen im Allgemeinen keine Auskunft über die Statistik der von ihnen verwalteten Daten geben, schlagen sich Änderungen an diesen Daten nicht automatisch in den zentralen Datenstatistikkatalogen nieder - die verwalteten Datenstatistiken werden obsolet. In der vorliegenden Arbeit wird die Erweiterung eines RDF-basierten Query-Federators beschrieben, die die Obsoleszenz von zentral verwalteten Datenstatistiken beurteilen und eine gegebenenfalls notwendige Aktualisierung einzelner Datenstatistiken unternehmen können soll. Als Grundlage dazu dient die Beobachtung auftretender Fehler in der Kardinalitätsschätzung ausgewerteter Queries. Eine Evaluation des Systems wird anschließend beschrieben. Die Ergebnisse zeigen die prinzipielle Richtigkeit der zugrundeliegenden Überlegungen, die praktische Anwendbarkeit kann jedoch nicht überzeugend demonstriert werden. Die Wiederverwendung der entwickelten Systemerweiterung für vielversprechendere Ansätze erscheint jedoch möglich und wird diskutiert.

Inhaltsverzeichnis

1 Einführung

In den zurückliegenden 10 Jahren hat im Forschungsbereich rund um webbasierte Informationssysteme eine Entwicklung zunehmend auf sich aufmerksam gemacht, die eine Fülle ambitionierter Zielen hat und auch verspricht diesen gerecht zu werden - die des Semantic Webs. Am Anfang dieser Entwicklung stand die Idee Information, die wie zuvor auch im World Wide Web (WWW) bereitgestellt wird, auf eine bestimmte Weise so strukturiert zu formulieren, dass auch Maschinen diese automatisch auslesen und logisch konsistent verknüpfen können. Auf diese Weise rücken seit jeher angestrebte Szenarien im Umgang mit Information zunehmend in greifbare Nähe. Ein Paradigma, das dies noch stärker betont, ist das der Linked (Open) Data (LOD), das seit 2006 intensiv verfolgt wird. Da man sich bei der Umsetzung von LOD gleichzeitig auf die Standards des Semantic Web stützt, kann man LOD folglich als einen speziellen Ausschnitt des Semantic Web selbst auffassen. Wie im Semantic Web auch, werden Daten nach dem LOD-Paradigma unter Verwendung des Resource Description Framework (RDF) ausgedrückt und lassen sich mit Hilfe von Abfragen in SPARQL erfragen (siehe Abschnitt 2.3). Im Unterschied zum Semantic Web werden für LOD jedoch explizit bestimmte Prinzipien propagiert und verfolgt, die über die Forderungen des Semantic Webs hinausgehen (siehe Abschnitt 2.2). Eine wichtige Konsequenz daraus ist die Eigenschaft, dass einzelne Datenbestandteile analog zum WWW über die Grenzen ihrer jeweiligen Domänen (z.B. der Datenquellen über die sie bereitgestellt werden) hinweg verknüpft werden können. Damit bilden sie in ihrer Gesamtheit einen zusammenhängenden Graphen, die sogenannte LOD-Cloud (vgl. 2.1). Dadurch lassen sich für eine Abfrage Brücken über gleich mehrere relevante Datenquellen schlagen, die Abfrage sich dadurch u.a. reichhaltiger beantworten. Jedoch, so fortschrittlich dieser Ansatz ist: Mögliche Vorteile erkauft man sich auf technologischer Seite jedoch auch mit einer im Allgemeinen deutlich erhöhten Komplexität bei der Verarbeitung der Query (siehe Abschnitt 2.4).

1.1 Motivation und Problemstellung

Die Tatsache, dass sich die Daten der LOD-Cloud auf potenziell sehr viele Quellen unterschiedlichen Umfangs verteilen, ist auf gewisse Weise also Se-

gen und Fluch zugleich. Möchte man Abfragen über diese verknüpfte Daten auswerten, stellt sich somit das Problem, dass für die erfolgreiche Auswertung zumindest für Teile der Query, gleich mehrere, möglicherweise sogar sehr viele, Datenquellen in Frage kommen. Eine mögliche Herangehensweise wird in föderierten RDF-Datenbanksystemen verfolgt. In einem solchen System sorgt eine Komponente, der Federator, dafür, dass eine Query, die an das System gestellt wird, möglichst gut in solche Teilanfragen zerlegt wird, die an geeignete Datenquellen delegiert werden können. Unter mehreren für eine (Teil-)Query in Betracht kommenden Datenquellen wird/werden im Idealfall diejenige(n) ausgewählt, die die meisten und/oder „aussagekräftigsten“ Ergebnisse liefert/liefern. Befähigt wird der Federator dazu durch statistische Informationen, die über die Datenquellen und deren Daten erhoben und gemeinsam mit den Adressen der Quellen in einem Katalog verwaltet werden. Auch die Query-Optimierung, die sich in der weiteren Query-Verarbeitung anschließt, benötigt diese Informationen, um auch optimale Auswertungspläne (QEP) herzustellen. In der (kostenbasierten) Optimierung werden die einzelnen QEP einer Query dazu auf den Umfang des von ihnen gelieferten Ergebnisses vorab abgeschätzt. Bei dieser Schätzung kommen dann ebendiese Statistiken über den Datenbeständen zum Tragen.

Die Qualität der Query-Fragmentierung und -delegation und die Query-Performance hängen also direkt von der Aktualität der zu verwendenden Statistiken und der Güte der Schätzungen über diesen ab. Sind diese schlecht, so werden möglicherweise auch schlechte QEP hergestellt. Die Teilabfragen, und damit die ursprüngliche Query insgesamt, werden dann nur unzureichend beantwortet bzw. nicht optimal ausgewertet. Aufgrund der inhärenten Datenunbeständigkeit innerhalb der LOD-Cloud insgesamt, ist es dann aber wichtig Statistiken und Schätzannahmen auch den tatsächlichen Datenbeständen der Datenquellen anzupassen, um eine möglichst gute Query-Auswertung stets zu gewährleisten. Da aber Veränderungen in einzelnen Datenquellen im Allgemeinen nicht angekündigt und auch nicht zeitlich vorhersehbar auftreten, ist es schwierig das immer für die einzelnen Datenquellen rechtzeitig zu erkennen. Für den Betrieb eines RDF-Federators aber wäre es von besonderem Interesse, veraltete Datenstatistiken bzw. ungenügende Schätzungen rechtzeitig und zuverlässig zu erkennen, um darauf ggf. kompensatorisch reagieren zu können.

1.2 Lösungsansätze

Man könnte versucht sein, z.B. die einzelnen SPARQL-Endpoints, die der Federator verwaltet, regelmäßig durch speziell erzeugte Abfragen auf statistische Informationen hin abzufragen. Dies mag in der Praxis im Einzelfall denkbar sein, ist allgemein jedoch nicht effizient durchführbar und daher als Lösungsansatz nicht unbedingt vielversprechend.

Eine Lösung des Problems könnte allerdings darin bestehen, im laufenden Betrieb eines RDF-basierten Datenbanksystems - bei Verarbeitung und Beantwortung der einzelnen Anfragen - explizite Beobachtungen anzustellen. Diese würden dann der automatischen Bewertung der Datenstatistiken und der Schätzungen über diesen dienen. Die „Messgröße“ dabei wäre beispielsweise die auftretende Abweichung des geschätzten vom tatsächlichen (Teil-)Ergebnisumfang, wie sie innerhalb eines erzeugten Auswertungsplans an mehreren Stellen zu beobachten sein sollte. Es existieren Arbeiten, die ähnliche Ansätze verfolgen (siehe Abschn. 3). So beschreiben [SLMK01] mit LEO (Learning Optimizer) einen lernenden Optimizer für das bekannte Datenbanksystem DB2. Dieses System leitet aus Abweichungen zwischen Kostenschätzung und -Realwert mit der Zeit Korrekturwerte ab, die bei folgenden Schätzungen kompensatorisch eingehen. Die Schätzungen werden somit im Laufe der Zeit immer besser. Ein anderes System wird in [MRS$^+$04] skizziert. In dem Paper wird ein progressiver Optimierungsvorgang (POP genannt) beschrieben, der Schätzfehler eines bestimmten Signifikanzniveaus als Auslöser für eine dynamische Re-Optimierung des gerade betrachteten Teil-QEP verwendet (siehe dazu 3).

Im Falle von LOD erschwert sich dies durch die Möglichkeit eine Query zu fragmentieren und in Teilen von mehreren Quellen beantworten zu lassen, also Föderation zu betreiben. Für die beiden genannten Systeme wurden zu diesem Zweck jeweils Erweiterungen für den Fall föderierter Datenbanksysteme beschrieben (siehe [EKMR06a], [EKMR06b]).

Die oben genannten Lösungen wurden für den Einsatz in klassischen, relationalen Datenbanken konzipiert. Im Unterschied dazu soll in der vorliegenden Arbeit ein Ansatz untersucht und verfolgt werden, der sich an einem gegebenen (föderierten) RDF-Datenbanksystem orientiert. Es werden dabei auch anfallende Schätzfehler innerhalb eines QEP beobachtet. Ziel soll es dabei sein, aus beobachteten Schätzfehlern mit der Zeit auf die Aktualität (bzw. einen Mangel davon) der Datenstatistiken zu schließen. Anfallende Indikationsmeldungen, die die Obsoleszenz einzelner Datenstatistiken anzeigen, sollen dabei erzeugt und geeignet ausgegeben werden können. Der Federator

könnte dann in der Folge zumindest mittelbar, d.h. durch das Zutun des Betreibers, unterstützt werden.

1.3 Aufbau der Arbeit

Zunächst wird im folgenden Kapitel 2 die begriffliche Grundlage für den Rest der Arbeit gegeben. Das betrifft Konzepte und Technologien des Semantic Web und LOD allgemein sowie der Query-Verarbeitung und -Optimierung im Speziellen.

In Kapitel 3 der Arbeit wird ein aktueller Überblick über verwandte Arbeiten vermittelt, die in der relevanten Literatur diskutiert werden. Eine Abgrenzung und etwaige Bezüge zum Beitrag der vorliegenden Arbeit werden klar herausgestellt.

Im Kapitel 4 wird das Konzept erläutert, das der zu entwickelnden Lösung zugrunde liegt. Die im Laufe der Arbeit entwickelten Verfahren und Algorithmen werden an dieser Stelle präsentiert. Bedingungen und Anforderungen die für die nachfolgende Implementierung von Bedeutung sind werden ebenfalls kenntlich gemacht.

Die Implementation dieses Konzepts, mit ihren technischen Eigenheiten, wird in Kapitel 5 der Arbeit detailliert dargestellt. Das umfasst softwaretechnische Entwürfe der Lösung sowie bestehende Systeme, auf die dabei zurückgegriffen wird.

Eine eingehende Untersuchung des zuvor beschriebenen Systems wird dann in Kapitel 6 vorgenommen. Dabei wird der Plan für die Evaluierung als erstes beschrieben. Die bei der Evaluierung gelieferten Ergebnisse werden dann detailliert wiedergegeben. Eine Bewertung dieser Ergebnisse wird im Anschluss in 6.5 separat vorgenommen.

Die Arbeit wird durch Kapitel 7 mit einem Fazit der vorangegangenen Kapitel geschlossen.

2 Grundlagen

Für die thematische Orientierung der vorliegenden Arbeit werden im Folgenden die wichtigsten Begriffe und Grundlagen dargestellt. Zunächst wird auf das Semantic Web und auf Linked Open Data als Einsatzgebiet (i.w.S.) eingegangen. Grundlegende Probleme und Herausforderungen, die sich dabei stellen, werden angesprochen. In den darauf folgenden Abschnitten wird dann ein direkter Bezug zur Arbeit hergestellt. Dazu werden solche konkrete Technologien einführend beschrieben, die über weite Teile der Arbeit von Bedeutung sind und die technische Einbettung für die hier beschriebene Lösung darstellen.

2.1 Begriffliche Abgrenzung

Um Missverständnisse im Umgang mit Begriffen in den folgenden Teilen von vornherein auszuschließen, ist es hilfreich sich zunächst Begriffe klarzumachen. In Teilen orientiert sich der Autor dabei an [Con97, S. 6-7]:

Informationssysteme Der Begriff Informationssystem ist vergleichsweise abstrakt und der Kommunikationstheorie entlehnt [Krc05]. Unter diesem Begriff fasst man ein komplexes System aus Hardware, Datenbank(en), Software, Information, Anwendungen und Benutzern zusammen. Er bildet also gewissermaßen den Querschnitt eines Benutzungsszenarios ab, in dem der Mensch (häufig in einer Expertenrolle) auf Information zugreift und mit Hilfe dieser z.B. eigene Entscheidungen herbeiführt.

Datenbank, Datenbankmanagementsystem, Datenbanksysteme Ein Datenbankmanagementsystem (DBMS) hat die Aufgabe einen zusammenhängenden logischen Datenbestand (eine Datenbank) zu verwalten. Dazu gehören u.a. folgende grundlegende Aufgaben:

Die Verarbeitung von Abfragen, die über die Schnittstellen des DBMS eingehen. Die besteht grob betrachtet im Übersetzen und Beantworten der Anfragen. Jede der Anfragen wird dazu übersetzt und nach logischen und physischen Gesichtspunkten optimiert. Danach wird sie auf einen Auswertungsplan abgebildet, der für die Beantwortung der Abfrage im letzten Schritt ausgeführt wird.

In der Transaktionsverwaltung werden sämtliche Abfragen und Veränderungsaufträge der Benutzer in atomare Transaktionen gekapselt. In der für den Benutzer transparenten Benutzungsdynamik können sich einzelne Transaktionen widersprechen. Das DBMS stellt durch Verwaltung der Transaktionen (u.a. durch Verwerfen einzelner Transaktionen) sicher, dass sich dies nicht auf den Datenbestand auswirkt. Erfolgreiche Transaktionen dürfen keine Inkonsistenzen hinterlassen und ihre Änderungen nicht verloren gehen.

Weitere Aufgaben eines DBMS bestehen in der Mehrbenutzersynchronisation, der Fehlerbehandlung mit Backup und Recovery und bei verteilten Datenbanken in der Integration der einzelnen Datenbestände.

Ein Datenbanksystem ist ein System, bestehend aus einem DBMS und einem (logischen) Datenbestand.

Datenquellen und Mediatoren Im Weiteren kann mit dem Begriff der Datenquelle ein URI-referenziertes Dokument, eine Datenbank, oder eine RDF-Serialisierung (RDF-Dump) gemeint sein. Er vermittelt also nur eine abstrakte Sicht auf eine (ggf. zugriffsoffene) logische Ansammlung von Daten. Für die Integration in komplexere Informationssysteme können Datenquellen durch spezielle Programme, sogenannte Wrapper, in eine Mediator-Wrapper-Architektur [Wie92] eingebunden werden. Dabei Stellen die Wrapper eine einheitliche Schnittstelle bereit, über die ein zentraler Mediator auf sie zugreifen kann[1]. Föderierten DBMS (siehe 2.5) liegt eine solche Architektur zugrunde. Die verteilten Datenquellen werden hier durch eigene DBMS verwaltet und über zugehörige Abfrageschnittstellen (z.B. sogenannte SPARQL-Endpoints) für die Delegation von Teilabfragen zugänglich gemacht.

2.2 Semantic Web und Linked Data

Das zentrale Ziel des Semantic Web ist es, Daten im World Wide Web (WWW) so zu formulieren, dass sie von Maschinen direkt und einheitlich gelesen, verarbeitet und verknüpft werden können. Das Problem an dem WWW wie es (u.a.) von Tim Berners-Lee 1989 erdacht und danach realisiert wurde (und weiter in Entstehung begriffen ist) ist, dass es im Wesentlichen nur den Mensch als einzigen Akteur vorsieht. Bei dem Versuch einzelne Websites und Dokumente im WWW Maschinen irgendwie zugänglich zu machen stellt sich

[1] Als Beispiel wird in [Bro10] ein Wrapper für die Einbindung XML-basierter Web-Datenquellen in Sesame-Umgebungen beschrieben.

im Grunde von Neuem die alte Herausforderung, Fließtext, der von Menschen für Menschen geschrieben wurde, maschinell zu „verstehen“. Die Herangehensweise, wie sie vom Semantic Web verfolgt wird, ist eine andere. Danach stellt man sich dieser Herausforderung nicht direkt, sondern umgeht sie vielmehr unter Verwendung eines abgestimmten technologischen Frameworks, durch das die präzise Semantik eines Inhalts nicht verloren geht. Dazu sind grundsätzlich drei gleichermaßen wichtige Dinge vonnöten. Zunächst müssen von Menschen in konsensualer Form gefundene Konsolidierungen von Begriffen und deren Beziehungen spezifiziert werden, die bezüglich bestimmter Anwendungsdomänen als gültig erachtet werden. Solche Spezifikationen müssen in Form von Vokabularen, Taxonomien oder Ontologien formal definiert und nicht zuletzt maschinenlesbar aufgeschrieben werden.

Nun müssen Daten, die später innerhalb ihrer zugehörigen Domäne erfragt und manipuliert werden sollen, mit Rückgriff auf solche Ontologien geeignet strukturiert verfasst werden. Dies geschieht so, dass das Schemawissen, das latent in Form von Ontologien vorliegt, auch bei der Verarbeitung der Daten hinzu gezogen werden kann/muss. Im Weiteren kann die verankerte Semantik nicht verloren gehen und die Beantwortung von Anfragen z.B durch Inferenz erheblich präzisiert und beschleunigt werden.

Zuletzt muss eine Menge standardisierter Werkzeuge vorgehalten werden, auf die ein Softwareentwickler zurückgreifen kann, um Informationsverarbeitung gemäß diesem Ansatz betreiben zu können. Sie unterstützen ihn bei der Serialisierung (dem Aufschreiben), der Verarbeitung, Abfrage, Kommunikation und Präsentation der Daten. Zudem dient die Standardisierung der notwendigen Interoperabilität in einem weltweiten webbasierten Einsatzgebiet für die vielen möglichen Anwendungen des Semantic Web.

Auf den ersten Blick bietet das Semantic Web damit nun einen grundlegend neuen Ansatz für die Bereitstellung, Verarbeitung und Abfrage von Information. Viel mehr jedoch propagiert die Bewegung rund um das Semantic Web Ziele, die für sich genommen z.T. bereits seit Jahrzehnten in unterschiedlichsten Forschungsdisziplinen verfolgt werden (siehe [HKRS08]). Der Erfolg der nun die Bestrebungen rund um das Semantic Web zeitigt, wurde jedoch erst durch den Rückgriff auf geeignete Technologien ermöglicht, die so erst in den letzten Jahren aufgekommen sind und von denen nicht wenige schon im Rahmen des WWW erprobt und zur Reife gebracht werden konnten.

Sämtliche dieser Standards werden im dem sogenannte Semantic Web Stack [Bra07] zu einer technologischen Architektur hierarchisch zusammengefasst.

Allerdings sind noch nicht alle Teile dieses Stacks vollständig realisiert und sind weiterhin in der Entwicklung begriffen. In Teilen stellt dieser Stack tatsächlich (und erkenntlich) auch eine Konsolidierung der technologischen Grundlage des WWW dar. So bilden Uniform Resource Identifier (URI)[BLFM05], die Unicode Zeichenkodierung[2] und die eXtensible Markup Language (XML) [BPSM+08] die untersten Ebenen dieses Stacks. Dabei dient XML als Austauschformat für das Resource Description Framework (RDF) [LS99], in dem Wissen über bestimmte Objekte, sogenannte Ressourcen, und deren Beziehungen ausgedrückt repräsentiert werden kann (siehe dazu Abschn. 2.3). Die einzelnen Ressourcen werden hier durch URI konzeptionell eindeutig abgebildet und bei der Serialisierung mit Hilfe des Namespace-Mechanismus von XML an spezifische Namensräume gebunden. Wissen auf der nächsthöheren Entwurfsebene, schematisches Wissen über die Domäne also, lässt sich in leichtgewichtiger Form durch RDF-Schema (RDFS) [BG04] ausdrücken. Komplexere Zusammenhänge auf Schemaebene lassen sich in Form von sogenannten Ontologien durch die Web Ontology Language (OWL)[WG09] ausformulieren. Aus Flexibilitätsgründen existieren genau genommen drei verschiedene Varianten von OWL, von denen der Benutzer nach Bedarf wählen kann. Diese unterscheiden sich vor Allem in ihrer Ausdruckskraft und damit in der Komplexität des Schlussfolgerns. Eine wichtige Rolle, die sich auch im weiteren Verlauf der Arbeit abzeichnen wird, kommt zudem der SPARQL Protocol and RDF Query Language (SPARQL) zu. In diesem Standard wird eine Sprache spezifiziert, mit Hilfe derer sich Abfragen von RDF-Daten ausdrücken lassen, die dazu an designierte Schnittstellen, sogenannte SPARQL-Endpoints, gerichtet werden (siehe Abschn. 2.3).

Das Semantic Web ist technologisch so ausgelegt, dass prinzipiell auch die Zusammenführung physisch und räumlich getrennter Datenquellen des Semantic Web „natürlich“ und für den Benutzer transparent ermöglicht werden kann. Dem gegenüber steht jedoch die Tatsache, dass Daten im Semantic Web de facto allzu oft zu großen Datensätzen gehören, die an spezifische und in sich geschlossene Anwendungsdomänen gebunden sind.

Unter diesem Gesichtspunkt lässt sich die Initiative hinter Linked Open Data (LOD) verstehen. Durch diese Bestrebung wird stärker auf eine Entwicklung analog zum WWW hingewirkt, so dass einzelne, öffentlich zugängliche Datenobjekte durch echte, d.h. auflösbare, Verknüpfungen über Domänengrenzen hinweg verbunden werden können. Es mag dies der Grund sein, warum Tim Berners-Lee selbst von LOD als dem „Semantic Web done right“ spricht (vgl. [BL08]). Semantic Web und LOD sind also keine entgegenge-

[2] `http://www.unicode.org/standard/standard.html`

setzten Konzepte. Vielmehr handelt es sich bei LOD um einen Ausschnitt des Semantic Web, bei dem weitere Einschränkungen, ausgedrückt in den Prinzipien des LOD, zum Tragen kommen.

Die Prinzipien, die Linked Data zugrunde liegen, lauten dabei wie folgt [BL08]:

1. Use URI as names for things.
2. Use HTTP URI so that people can look up those names.
3. When someone looks up a URI, provide useful information, using the standards (RDF, SPARQL).
4. Include links to other URI, so that they can discover more things.

Einzelne Datenquellen, die selbst eine Menge von Daten z.B. über SPARQL zugänglich machen, können also eigenverantwortlich über HTTP URI relevante Verknüpfungen zu anderen Datenquellen veröffentlichen, die ggf. weiterführende Information verwalten. Insgesamt wird dadurch ein Geflecht aus vielen Datenquellen gebildet - die sogenannte LOD-Cloud (siehe Abb. 2.1).

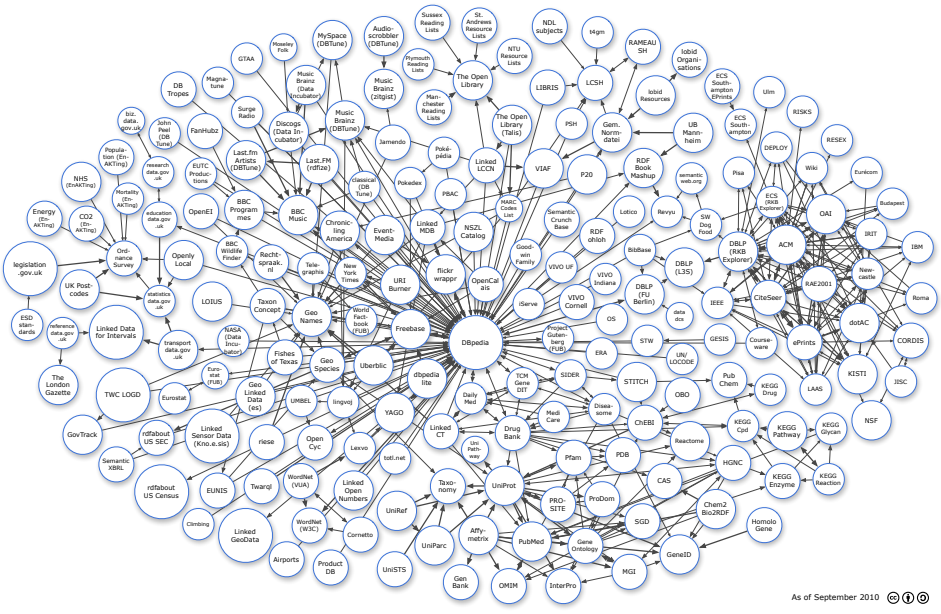


Abbildung 2.1: Linking Open Data cloud diagram, zusammengestellt von Richard Cyganiak und Anja Jentzsch (`http://lod-cloud.net/`).

2.3 RDF und SPARQL

Der für Semantic Web und LOD wahrscheinlich prägendste und auch wichtigste Standard aus dem Semantic Web Stack ist das Resource Description Framework (RDF) [LS99]. Mit RDF lassen sich beschreibende Daten, sogenannte Metadaten, über eindeutig identifizierte Ressourcen (wie z.B. Webdokumente) ausdrücken. Dazu gehören auch Beziehungen zwischen solchen Ressourcen, die sich ebenfalls abbilden lassen. Wichtig ist, dass es keinen Unterschied macht, ob das Beschriebene eine reale Verkörperung (beispielsweise in Form von auf Servern gelagerten HTML-Dokumenten) besitzt oder ob es rein konzeptioneller Natur ist. Somit lässt sich unter Verwendung von RDF auch sogenanntes schematisches Wissen bezüglich einzelner Anwendungsdomänen repräsentieren. Für das Semantic Web erweist sich RDF dadurch insgesamt als grundlegender technologischer Baustein. Erst mit Hilfe dieser Technik lassen sich Daten des Semantic Web so geeignet strukturieren, dass sie für die weitere Semantik-orientierte Verarbeitung in dem Rahmen des Semantic Web auch zugänglich sind.

Um über einzelne Ressourcen und deren Eigenschaften explizite Aussagen zu beschreiben, werden diese in RDF mit Hilfe von URI repräsentiert. Diese Bezeichner sind dadurch auch an bestimmte Namensräume gebunden. Zudem können sie auch aus formalen Ontologien stammen, in denen schematische Zusammenhänge spezifiziert und auf diese Weise spezifische Vokabulare in je einem Namensraum verankert werden. Im Übrigen lassen sich bereits auf der Ebene von RDF sehr einfache schematische Informationen ausdrücken. Dazu bietet RDF auch ein eigenes Vokabular aus URI-Bezeichnern, mit dessen Hilfe beispielsweise Typisierungen für Datenwerte ausgedrückt werden können. Dieses RDF-Vokabular wird durch RDF-Schema so erweitert, dass auch ohne Sprachen wie OWL zumindest einfache Ontologien auf Basis von RDF ausgedrückt werden können. Das umfasst beispielsweise Klassenhierarchien und -Instanziierungen.

Das Datenmodell des RDF (RDF-Modell) ist für sich betrachtet sehr einfach. Die kleinste und zugleich zentrale Einheit dieses Datenmodells ist die Ressource. Wie gesagt kann es sich bei einer solchen Ressource sowohl um ein real existierendes Ding als auch um ein abstraktes Konzept handeln, das sich nur dem menschlichen Verstand erschließt. Entscheidend ist, dass sie innerhalb des RDF-Modells durch URI global eindeutig und Namensraumgebunden identifiziert werden kann.

Eine auf diese Weise adressierbare Ressource kann mit einer zweiten Ressource oder einem Datenwert, einem Literal, eine Beziehung eingehen. Dabei

haben diese Beziehungen eine Richtung, weshalb es sich anbietet an Eigenschaften einzelner Ressourcen zu denken. Eine solche Eigenschaft wird in RDF durch sogenannte RDF-Tripel (bzw. sog. Triple-Statements) spezifiziert und repräsentiert. Ein Tripel dieser Art ist ein elementarer Datensatz in dem drei Bestandteile geordnet zusammengefasst werden. An erster Stelle steht eine Ressource, das Subjekt (engl. subject). An zweiter Stelle folgt das Prädikat (engl. predicate), das erst die eigentliche Eigenschaft (engl. property) benennt. Zuletzt folgt darauf das Objekt (engl. object). Bei dem Objekt kann es sich dabei ebenfalls um eine Ressource oder ein (ggf. typisiertes) Literal handeln. Eine Besonderheit stellen anonyme Ressourcen, die sog. Blank-Nodes, dar, die anstelle von Subjekt oder Objekt auftreten können und nicht durch globale URI sondern nur durch lokale ID repräsentiert sind. Diese dienen der Gruppierung spezifisch zusammengehöriger Eigenschaften einer (nicht-anonymen) Ressource, indem sie diese an sich binden und selbst als Objekt an die jeweilige Ressource angeknüpft werden. Neben dem Subjekt wird auch das Prädikat durch einen URI bezeichnet. Da in RDF alles eine Ressource ist, was durch einen URI abgebildet ist, kann auch das Prädikat genau genommen als Ressource aufgefasst werden. Eine Tatsache die bei der Reifizierung von Triple-Statements zum Tragen kommt. Schließlich kann auch anstelle des Objekts ein URI treten, sofern es sich bei der spezifizierten Beziehung um eine solche handelt, die zwischen zwei Ressourcen besteht. Ansonsten steht hier ein Datenwert, der, falls nicht durch eine Typangabe annotiert, wie ein String behandelt wird.

Das umfassende Konzept, das dem RDF-Modell zugrunde liegt, ist das des Graphen. Eine beliebige Menge von RDF-Tripeln bildet demnach einen (gerichteten) Graph, dessen Knoten Ressourcen oder Literale sind. Diese Knoten sind durch gerichtete Kanten, die Prädikate einzelner Tripel, untereinander verknüpft. Diese vereinende graphenorientierte Sicht bezüglich RDF verdeutlicht die Möglichkeit der Datenintegration. So macht es keinen Unterschied, ob RDF-Daten in einem zusammenhängenden Datensatz oder auf irgendeine Weise räumlich separiert vorliegen. In beiden Fällen lassen sie sich bei gegebenem Zugriff auf Basis des RDF-Modells vergleichsweise einfach als ein Graph, und damit als ein zusammengehöriger (logischer) Datenbestand, interpretieren, zusammenführen und verarbeiten. Problematisch bei der Datenintegration sind ggf. jedoch die „Kosten“ einer solchen Vereinigung, die im praktischen Einsatz natürlich von hervorgehobener Bedeutung sind (siehe 2.5).

Für die textuelle Beschreibung eines RDF-Graphen auf Basis einer konkreten Syntax, also für die Serialiserung des Graphen, bieten sich verschie-

denen Sprachen an. Eine sehr wichtige Sprache, die direkt auf XML beruht, ist RDF/XML [BM04]. Diese hat jedoch den Nachteil, für Menschen nicht sehr einfach zu lesen zu sein. Einfacher und konziser sind Serialisierungen auf Basis von Notation3 (N3) [BL01]. Dazu gehören auch die beliebten N3-Subsprachen Turtle [BBL11] und N-Triples [BB06]. Turtle und N3 bieten verschiedene Sprachkonstrukte, die insgesamt zu kompakteren Beschreibungen und damit der Lesbarkeit dienen. Dazu zählen z.B. Präfixdeklarationen für Namensräume, die kompakte Beschreibung sternförmiger Teilgraphen durch Weglassung gleicher Subjekt- (und ggf. Prädikats-)Bezeichner und Gruppierungen durch Klammerung anstelle von Blank-Nodes. Die Syntax von N-Triples, schließlich, ist die denkbar einfachste. Hier werden die Bezeichner der Tripelbestandteile einzeln geklammert und in ihrer Reihenfolge hintereinander aufgeschrieben, einzelne Tripel-Statements dabei durch „.“ abgeschlossen. Dokumente in N-Triples findet man häufig bei der Darstellung größerer Datensätze, zumal sie besonderes leicht zu verarbeiten sind.

Einzelne RDF-Datensätze oder -Graphen werden in der Praxis in eigens dafür geschaffenen Datenbanksystemen, sogenannten Triplestores, vorgehalten und verwaltet. Gängige Triplestore-Systeme setzen selbst häufig auf verschiedenen existierenden Speichermechanismen auf. Im Einzelfall können dabei durchaus relationale Datenbanken gekapselt werden, in denen die Datensätze letztlich eingelagert werden. Für die Abfrage und Manipulation der Datenbestände besteht dadurch ggf. zwar die Möglichkeit, beispielsweise über SQL-Anfragen, einen direkten Zugriff auf einzelne Daten zu erhalten. Dies ist jedoch z.B. dann nicht (direkt) möglich, wenn Datensätze des Triplestores stattdessen in serialisierter Form in eigenen Dateien vorliegen oder ausschließlich im Hauptspeicher des Systems existieren. In jedem Fall entfernt man sich dadurch aber von dem RDF-Datenmodell, in dem die Daten ja eigentlich eingefasst sind.

Um diesen Problemen zu begegnen, wurde durch das W3C in der Zwischenzeit mit SPARQL (SPARQL Protocol And RDF Query Language) der wichtigste Standard für eine einheitliche Abfragesprache auf Basis des RDF-Modells verabschiedet[PS08]. Mit SPARQL wird neben der angesprochenen Abfragesprache (SPARQL Query Language for RDF) noch ein Protokoll (SPARQL Protocol) spezifiziert, das den tatsächlichen Kommunikationsablauf zwischen abfragenden Clients und der SPARQL-Service-Instanz (SPARQL-Endpoint) des Triplestores regelt. Ein solcher SPARQL-Endpoint kapselt die im Triplestore verwendeten Datenbanksysteme und Speichermechanismen nach außen durch ihre einheitliche Schnittstelle ab. Seit einigen

Jahren wird SPARQL von den wichtigsten gängigen Triplestore-Systemen bevorzugt unterstützt und ist demzufolge auch für die vorliegende Arbeit von Bedeutung.

Die SPARQL-Abfragesprache basiert wie gesagt auf dem RDF-Modell und ist damit ebenfalls graphbasiert. Zu diesem Aspekt gehört das Konzept des Graphmusters (engl. graph pattern). In einer konkreten Abfrage wird immer ein solches Graphmuster spezifiziert, das für die Beantwortung der Abfrage am Datenbestand des Triplestores ausgewertet wird. Die Syntax der SPARQL-Anfragesprache ähnelt dabei der von Turtle. Eine typische Abfrage in SPARQL hat die folgende syntaktische Form:

```
PREFIX owl: <http://www.w3.org/2002/07/owl\#>
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
PREFIX lmdb: <http://data.linkedmdb.org/resource/movie/>
PREFIX dbpedia <http://dbpedia.org/>

SELECT ?page ?abstract ?performance WHERE {
   ?x    lmdb:actor_name   'Peter Sellers';
         lmdb:performance  ?performance.
   ?x    owl:sameAs        ?y.
   ?y    foaf:page         ?page;
         dbpedia:ontology/abstract ?abstract.
}
```

Abbildung 2.2: Beispiel einer verteilten SPARQL-Query: Gesucht werden alle Filmauftritte von Peter Sellers, die bei LinkedMDB.org eingetragen sind. Zusätzlich wird von DBPedia.org eine Kurzbeschreibung zu seiner Person und ein Link zur zugehörigen Wikipedia-Seite abgefragt. Die Verknüpfung zwischen den beiden Datenquellen findet über ein „sameAs“-Tripel statt, das aus einer dritten Datenquelle (Example.com) stammt.

Durch `PREFIX` werden im Kopf der Abfrage jeweils einzelne Namensraumabkürzungen eingeführt, die im folgenden Teil verwendet werden können. `SELECT` bestimmt die Liste aller Variablen, die in der Abfrage vorkommen und ausgegeben werden sollen (alternativ ist die Angabe „*“ möglich, womit die vollständige Variablenliste gemeint ist). Im `WHERE`-Block wird schließlich das Graphmuster der Abfrage beschrieben. Bei der Auswertung eines solchen Graphmusters wird versucht, dieses mit dem Datenbestand des Triplestores in Übereinstimmung zu bringen. Sind Übereinstimmungen, d.h. auf das Muster passende Teilgraphen aus dem Datenbestand, vorhanden, so werden dieses für die Beantwortung der Abfrage herangezogen.

Mehrere einzelne Graphmuster lassen sich zu größeren Mustern kombinieren und bestehen umgekehrt aus einzelnen sogenannter Triplepatterns (Tripel-Muster). Diese elementaren Graphmuster sind Muster, die mit einzelnen Tripel-Statements aus dem Datenbestand abzugleichen sind, und haben eine Struktur analog zu der eines Triple-Statements. Im Unterschied zu einfachen Tripel-Statements dürfen jedoch anstelle von Subjekt, Prädikat oder Objekt auch Variablen stehen. Ein Triplepattern hat dann Übereinstimmungen im Datenbestand, wenn es Tripel gibt, die an den Stellen, an denen im Muster keine Variablen stehen, jeweils denselben RDF-Term (URI, Literal oder Blank-Node) aufweisen. Etwaige Variablen im Muster werden dann mit den zugehörigen Werten der passenden Tripel besetzt. Jede Variablenbelegung, die so durch ein übereinstimmendes Triple gebildet wird, bezeichnet man als Lösung des Triplepatterns. Hat ein Tripel mehrere Übereinstimmungen, werden die einzelnen Lösungen in einer Lösungssequenz zusammengetragen und zur weiteren Auswertung der Query als Teilergebnis übergeben.

Mehrere Triplepattern können zunächst ein sogenanntes einfaches Graphmuster (engl. basic graph pattern, BGP) bilden. Für die Auswertung eines BGP werden die Lösungssequenzen der Triplepattern des BGP gebildet und durch Verbundoperationen, analog zum Join-Operator der Relationalen Algebra, zusammengefasst. Dazu kann man sich die Lösungssequenzen als Relationen vorstellen, deren Tabellenschemata durch die Variablenbezeichner der zugehörigen Triplepatterns gebildet werden. Das Tabellenschema für die Lösungssequenz eines BGP besteht somit aus den Bezeichnern aller irgendwo im BGP vorkommenden Variablen.

Neben einfachen Graphmustern existieren in SPARQL auch komplexe Graphmuster, für die unter Anwendung bestimmter Operatoren mehrere BGP kombiniert werden können. Zu diesen Operatoren zählen `UNION` und `OPTIONAL`. Diese sind jeweils linksassoziativ und verrechnen ein linksstehendes (komplexes) Graphmuster mit einem rechtsstehenden BGP. Durch `UNION` werden alternative Graphmuster deklariert, die insgesamt letztlich durch die Vereinigung der zugehörigen Lösungssequenzen realisiert werden. Mit `OPTIONAL` werden in der Anfrage optionale Graphmuster angegeben. Diese müssen dabei rechts stehen und werden nur dann in die Rechnung mit einbezogen, wenn mindestens eine Übereinstimmung vorhanden ist.

Zur Syntax der SPARQL-Abfragesprache gehören noch viele weiter Ausdrucksmittel. Dazu gehören Angaben von Filterbedingungen, die sich an BGP anknüpfen lassen, die von allen Lösungen des Ergebnisses erfüllt werden müssen. Zudem existieren Modifikatoren, über die die Reihenfolge und der Umfang einer Lösungssequenz beeinflusst werden können, z.B. durch Sor-

tierung nach bestimmten Attributen oder durch Filterung nach gegebenen Bedingungen. Auch für die Formatierung der Lösungssequenzen existieren Mittel. Dadurch lassen sich mit Hilfe der Lösungen vollkommen neue Ergebnisgraphen komponieren, die in Form eines RDF-Graphen zurückgeliefert werden.

Der Abfragenstandard von SPARQL definiert auch eine Algebra, über die die operationale Semantik der Abfragesprache spezifiziert wird. Diese erinnert im Kern an die von Codd eingeführte Relationale Algebra [Cod70] und eignet sich im Weiteren für eine abstrakte Betrachtung von SPARQL-Abfragen und deren Verarbeitung. Dazu ist es notwendig die Abbildung der verschiedenen Sprachkonstrukte auf die einzelnen Operatoren ebendieser Algebra zu kennen.

Die einzelnen Operatoren der SPARQL-Algebra werden auf Lösungssequenzen angewendet, wie sie oben bereits angesprochen wurden. Bei der Auswertung eines Operators wird in Abhängigkeit von möglichen weiteren Operatorargumenten eine weitere Lösungssequenz errechnet, deren Elemente (je nach physischer Implementierung) Pipeline-artig sukzessive ausgegeben werden können. Insgesamt lassen sich somit mehrerer solcher Operatoren bzw. deren Anwendungen verketten. Da es in der SPARQL-Algebra neben unären auch binäre Operatoren gibt, bildet eine konkrete Operatorverkettung, angewendet auf mehrere Eingabesequenzen, eine baumförmige Struktur aus, wie sie in Abbildung 2.3 zu sehen ist.

Die wichtigsten Operatoren der SPARQL-Algebra sind im Einzelnen wie folgt [LS99]:

BGP repräsentiert ein einfaches Graphmuster innerhalb einer Query und hat als Argument eine Liste von Triple-Patterns, die direkt an dem RDF-Graphen zur Übereinstimung gebracht werden müssen. Die ersten Teilergebnisse werden innerhalb eines SPARQL-Plans also durch BGP-Operatoren gebildet.

Filter erwartet einen logischen Filterausdruck und eine Lösungsequenz als Arguemente. Die Elemente der Lösungsequenz werden nur gemäß der Filterbedingung in die resultierende Lösungsequenz aufgenommen.

Join erwartet als Argument zwei Lösungsequenzen, die von zugehörigen Kind-Operatoren gebildet werden. Die Argumentesequenzen werden gemäß eines natürlichen Verbunds der relationalen Algebra zusammengeführt.

LeftJoin repräsentiert die Auswertung optionaler Graphmuster und erwartet zwei Lösungsequenzen und einen Filterausdruck als Argumente.

Der Filterausdruck ist zunächst *true*, entspricht aber ggf. dem Filterausdruck, der in dem durch ein OPTIONAL eingeführten Graphmuster einer Query spezifiziert wird. Die „rechte“ Lösungssequenz entsteht zuvor bei der Auswertung des optionalen Graphmusters. Die Auswertung eines LeftJoin entspricht dem linken äußeren Join der relationalen Algebra.

Union erwartet zwei Lösungssequenzen, berechnet von zwei Kindoperatoren, als Argumente. Diese werden zu einer Lösungssequenz vereinigt.

OrderBy modifiziert eine übergebene Lösungssequenz für die Ausgabe, indem die Elemente nach einer gegebenen Liste von Sortieranweisungen sortiert werden.

Project modifiziert eine übergebene Lösungssequenz für die Ausgabe, indem aus deren Elementen nur die Belegungen für eine ebenfalls bestimmte Menge von Variablen übernommen werden.

Distinct modifiziert eine übergebene Lösungssequenz für die Ausgabe, indem duplizierte Elemente (bzgl. der Variablenbelegung) entfernt werden.

Dieses Modell lässt sich weiter verfeinert betrachten. Anstelle des BGP-Operators lässt sich durch einen neuen Operator ebenso eine algebraische Entsprechung für einzelne Triplepattern einführen. Eine konkrete Anwendung dieses Operators liefert dann die zugehörige Lösungssequenz eines Triplepatterns einer Anfrage. Damit lässt sich eine einzelne Anwendung des BGP-Operators durch (im Allgemeinen) mehrere Anwendungen dieses Operators[3] ersetzen. Die dabei gelieferten Lösungssequenzen müssen dann nur durch Anwendungen von Join zusammengeführt werden. Eine Visualisierung der Anfrage aus 2.2 gemäß dieser Festlegung findet sich in Abbildung 2.3.

2.4 Query Processing und Optimierung

Der Sinn eines Datenbanksystems besteht, wie bereits erwähnt, zunächst darin, von den Eigenheiten der Datenhaushaltung und -Verarbeitung (und den resultierenden Problemen) weitestgehend zu abstrahieren. Dazu muss das DBMS eine ganze Reihe von Aufgaben übernehmen, die sicherstellen, dass die Datenhaushaltung aus Anwendersicht transparent bleibt. Eine zentrale

[3] Als Beispiel sieht das algebraische Modell von Sesame hierfür einen `StatementPattern`-Operator vor.

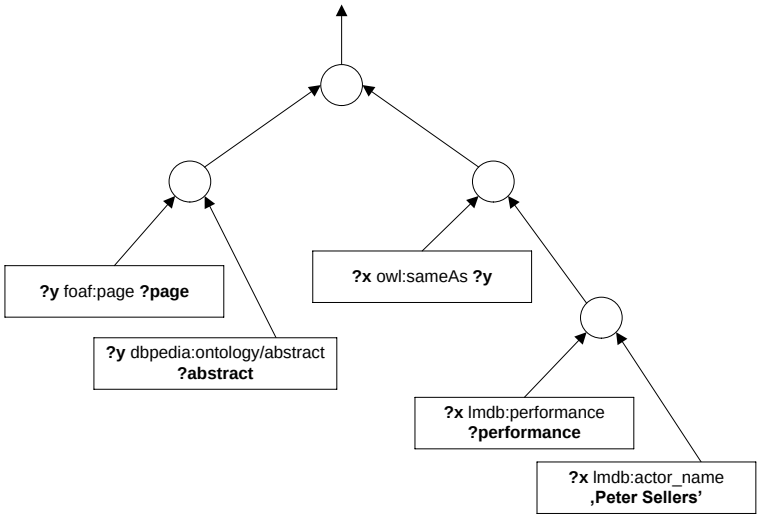


Abbildung 2.3: Abstrakter algebraischer Query-Ausdruck, der auf die SPARQL-Anfrage aus 2.2 passt. Die durch Kreise angedeuteten Knoten sind hier binäre Join-Operatoren.

Aufgabe daraus ist die Abfrageverarbeitung (engl. query processing). Moderne Datenbanksysteme stellen ihre Dienste über eine Schnittstelle gegenüber Anwendungen und Benutzern zur Verfügung, deren wichtigstes Kommunikationsmittel die Abfrage ist. Eine Abfrage (engl. query) ist ein Sprachausdruck über einer deklarativen Abfragesprache, wie z.B. SQL bei relationalen Datenbanksystemen oder SPARQL im Semantic Web. In einer Abfrage werden Daten einer Datenbank abgefragt, indem bestimmte Anforderungen an die Ergebnismenge spezifiziert werden. Jedes Element der schließlich gelieferten Ergebnismenge muss diesen Anforderungen also genügen. Da durch moderne Abfragesprachen also nur spezifiziert wird *was* zurückgegeben werden soll, nicht aber *wie*, muss aus einer Query ein Auswertungsplan (query execution plans, kurz *QEP*) erzeugt werden, der äquivalent zur Abfrage ist und festlegt auf welche Weise die Abfrage auszuwerten ist. Für eine formulierte Query existieren daher im Allgemeinen sehr viele dieser operationalen Ausprägungen [Ioa96]. Diese können alle der Beantwortung derselben gestellten Abfrage dienen, unterscheiden sich untereinander je aber (z.T.) erheblich in dem notwendigen Rechenaufwand, etwaigen Übertragungskosten und insgesamt der Antwortzeit. Das bedeutet, dass für eine Query immer ein möglichst guter, idealerweise der optimale, QEP zu suchen ist.

Aus der einschlägigen Literatur lässt sich ein für klassische Datenbanksysteme charakteristischer Ablauf des Query Processings identifizieren (siehe dazu [KE06, JK84, Mit95, Ioa96]). Darin wird zunächst die, von einem Benutzer

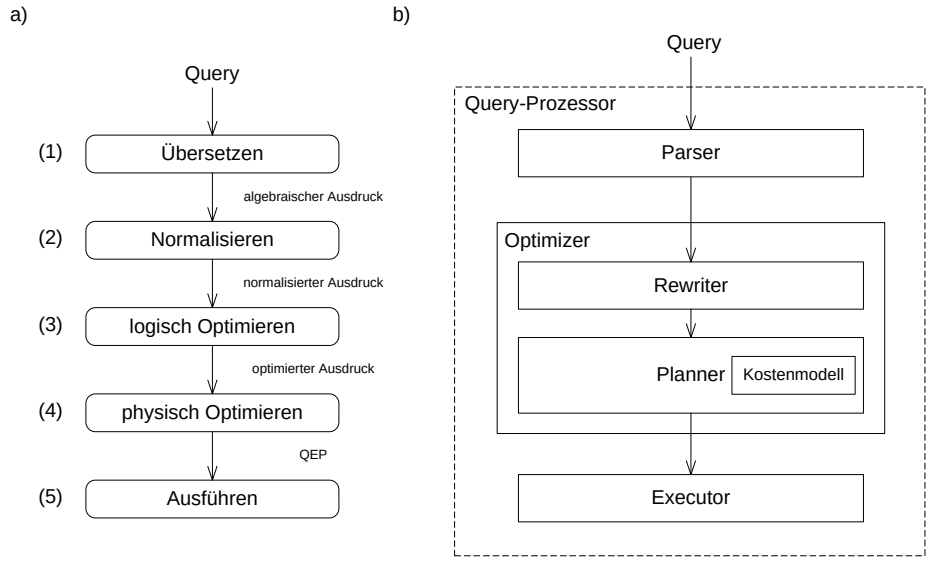


Abbildung 2.4: a) klassischer Ablauf des Query Processings, b) zugehörige Grobarchitektur eines Query-Prozessors.

manuell, oder halbautomatisch aus Vorlagen erzeugte Query an einen Server des adressierten Datenbanksystems übergeben und dort entgegengenommen. Die Query wird seitens des DBMS dann in mehreren Schritten verarbeitet:

Schritt 1: Die empfangene Query wird gelesen, analysiert und abhängig von der Abfragesprache aus der textuellen Form in einen äquivalenten algebraischen Ausdruck übersetzt.

Schritt 2: Der Ausdruck wird transformiert. Zunächst wird der Ausdruck in eine Ausgangsform standardisiert.

Schritt 3: Der Ausdruck wird durch Anwendung von Transformationsregeln vereinfacht und heuristisch optimiert.

Schritt 4: Für den (logisch) optimierten Query-Ausdruck wird eine Menge alternativer QEP erzeugt. In einer physischen Optimierung werden die alternativen QEP auf Basis eines Kostenmodells in ihren Kosten abgeschätzt und ein als optimal erkannter QEP ausgegeben.

Schritt 5: Der „optimale“ QEP wird ggf. in einen ausführbaren Code übersetzt, ausgeführt und die Query letztlich durch die zurückgegeben Ergebnisse beantwortet.

Die obige Phaseneinteilung schlägt sich auch entsprechend in der Komponentensicht der abstrahierten DBMS-Architektur nieder (siehe Abb. 2.4 b)).

Eine von einem DBMS entgegengenommene und durch einen Parser übersetzte Query wird einem Optimizer übergeben, der sich selbst in zwei Subkomponenten einteilen lässt. Ein Rewriter nimmt die geparste Query entgegen und führt darauf die logischen Transformationen aus Schritt 2 aus. Die Schritte 3 und 4 übernimmt anschließend ein Planner, dem dazu u.a. Kosten- und Kardinalitätsschätzer angeschlossen sind. Für die vorliegende Arbeit liegt der Fokus auf Schritt 4, der physischen Optimierung der Queries. Ein QEP ist ein Ausdruck über einer physischen Algebra, deren Operatoren durch den (physischen) Optimizer abschätzbar sind. Das dazu herangezogene Kostenmodell ist systemspezifisch und wird durch statistische Informationen bezüglich Systemleistung und Datenverteilung unterfüttert. Auf dieser Grundlage lassen sich durch Schätzungen der Kardinalitäten bzw. Selektivitäten der einzelnen Teilausdrücke eines QEP sowohl der zu erwartende Ergebnisumfang als auch die Auswertungskosten abschätzen. Das Optimierungsproblem, also die Suche nach dem günstigsten QEP, ist dabei außerordentlich schwer - bereits die Feststellung einer optimalen Join-Reihenfolge ist NP-hard [IK84]. Nur aufgrund der Tatsache, dass Queries zumeist nur wenige Operatoren beinhalten, lassen sich optimale QEP in annehmbarer Zeit finden. Als Suchverfahren kommt dafür in den meisten Fällen Dynamic Programming zum Einsatz [Ioa96].

Ein Problem, das bei der kostenbasierten Query-Optimierung grundsätzlich besteht ist, dass die im Hintergrund vorgehaltenen Statistiken selbst nur grobe Schätzungen darstellen oder unter Berücksichtigung bestimmter Annahmen nur grobe Schätzungen erlauben. Das liegt im Wesentlichen daran, dass man als Betreiber einer Datenbank zumeist der Notwendigkeit eines Trade-Off, eines Kompromisses, zwischen der Genauigkeit und Aussagekraft der Statistiken und deren Umfang und Verwaltungsoverhead unterworfen ist. Problematisch sind auch allzu häufige Datenänderungen in der jeweiligen Datenbank. Statistiken über den Datenbeständen verlieren dabei mitunter ihre Gültigkeit. Schätzfehler sind dann unvermeidlich - unabhängig von den verwendeten Schätzverfahren. In der Praxis kommt es also zwangsläufig zu Abweichungen zwischen der Schätzung und dem Realumfang eines Anfragenergebnisses, die somit die Erzeugung optimaler QEP mitunter nur näherungsweise erlauben.

2.5 Triplestores und Föderation

Mit dem Begriff des Triplestores, der vergleichsweise weit ausgelegt wird, werden bestimmte Speichersysteme für das Semantic Web bezeichnet. Triplestores fassen gleich mehrere Ebenen des Semantic-Web-Stack integrativ zusammen und stellen damit einen wichtigen technologischen Baustein für die Infrastruktur des Semantic Web - und damit auch für Linked Data - dar. Die grundsätzliche Aufgabe dieser Systeme besteht in der Aufbewahrung von RDF-Tripeln, sowie der Bereitstellung von Abfrage- und Manipulationsmechanismen [Rus]. Im Allgemeinen handelt es sich also um Datenbanksysteme, die speziell auf den Einsatz im Umfeld des Semantic Web ausgerichtet sind. Im Unterschied zu ihren „klassischen“ relationalen Pendants, basieren diese aber nicht (direkt) auf dem Relationalen Modell nach Codd ([Cod70]), sondern auf dem RDF-Datenmodell. Allerdings können bei gängigen Triplestore-Systemen, neben der direkten Hauptspeicher- und/oder Festplattennutzung, in den meisten Fällen auch unterstellte (relationale) Datenbanksysteme für die physische Aufbewahrung des Datenbestandes herangezogen werden.

Zu den ersten Aufgaben zählt an dieser Stelle die direkte Manipulation des Datenbestandes, d.h. das Laden, Verändern und Löschen von Tripeln auf RDF-Ebene. Dazu werden komfortable Schnittstellen angeboten, über die sich RDF-Serialisierungen in verschiedenen Formaten laden lassen. Moderne Triplestores stellen zudem Schnittstellen nach Außen zur Verfügung, über die RDF-basierte, deklarative Abfragen (z.B.) in SPARQL entgegengenommen und beantwortet werden können. Dadurch können sie im Übrigen auch als Quellen für Linked Data (z.B. auch innerhalb der LOD-Cloud) fungieren. Eine wichtige Aufgabe eines Triplestores ist zudem die Inferenz. Damit lassen sich bei der Beantwortung von Anfragen ergänzend logische Schlussfolgern beisteuern, die sich durch Räsonieren über vorliegenden ontologisch abgebildeten Zusammenhängen ergeben können.

Im praktischen Umgang mit Triplestores müssen diese darüber hinaus natürlich denselben Ansprüchen genügen wie andere klassische Datenbanksysteme. Sie müssen damit also auch bestimmte Aufgaben übernehmen, die für den reibungslosen Ablauf unabdingbar sind und ebenso in den Verantwortungsbereich eines Datenbankmanagementsystems (DBMS) fallen. Dazu gehören, neben dem Datenzugang, eine Transaktionsverwaltung, die Mehrbenutzersynchronisation sowie die Gewährleistung von Datensicherheit und -Integrität. Von hervorgehobenem Interesse bezüglich der vorliegenden Arbeit ist jedoch vor Allem die Verarbeitung eingehender Anfragen, einschließlich der Optimierung anhand von Kostenmodellen (siehe Abschn. 2.4). Zu den ak-

tuell populärsten Triplestore-Systemen lassen sich (u.a.) vor Allem Sesame[4], Jena[5], Joseki[6], 3Store[7] und AllegroGraph[8] zählen.

Für die Beantwortung von Abfragen über Linked Data existieren mehrere mögliche Ansätze, die in der einschlägigen Forschung auch bereits sehr genau studiert worden sind. Bei dem aktuell womöglich gängigsten Ansatz, dem sogenannten Datawarehousing, werden RDF-Daten all jener Datensätze zentral gesammelt und indiziert, die für sämtliche der folgenden Abfragen zur Verfügung stehen sollen. Diese Abfragen lassen sich dann in der Regel sehr effizient und mit kleinen Antwortzeiten auswerten. Gerade bezogen auf Linked Data ist dieser Ansatz jedoch nicht immer sinnvoll. Neben der u.U. unnötig verursachten Redundanz, unterminiert dieser Ansatz die Ziele, die durch die Linked-Data-Prinzipien erst ins Auge gefasst werden. Zudem sprechen weitere Einwände, die weiter unten angesprochen werden, gegen Datawarehousing. Neuer und direkt an den Linked-Data-Prinzipien orientiert ist ein explorativer Ansatz für die Beantwortung von Linked-Data-Abfragen. Hier wird die Query an einem dynamisch und inkrementell wachsenden Datenbestand ausgewertet. Gebildet wird dieser Datenbestand aus den Daten, die durch die Dereferenzierung von URI auffindbar sind. Angefangen bei den URI, die bereits in einer Query enthalten sind, wird der Datenbestand durch Dereferenzierung aller zwischenzeitlich in die Lösungsmenge der Query aufgenommenen URI ergänzt. Während der Auswertung der Query wird eine wachsende Hülle referenzierter RDF-Datenquellen gebildet. Sobald die Hülle der besuchten Quellen nicht weiter wächst, die Query also an keiner weiteren referenzierten Datenquelle ergänzend ausgewertet werden kann, ist der Verarbeitungsvorgang abgeschlossen. Ein solches Verfahren wird in [HBF09] verfolgt und birgt den Vorteil, dass damit eine beliebige Query über Linked Data ad hoc ausgewertet werden kann. Jedoch besteht zum einen der Nachteile, dass Queries möglicherweise nur an wenigen und/oder unzureichenden Datenquellen ausgewertet und damit aller Erwartung nach unvollständig beantwortet werden. Die Auswahl der Quellen geschieht mechanisch und ist nicht direkt an der „tatsächlichen Eignung“ der Quellen orientiert. Zum anderen wächst die Hülle nur entlang bereits im Datenbestand enthaltener Links. Beinhaltet eine nicht in die Hülle aufgenommene, aber u.U. geeignete Quelle einen Link hinein in die Hülle, so wird diese Quelle dennoch nur dann für die Beantwortung der Query in Betracht gezogen, wenn ein zweiter Link

[4] `http://www.openrdf.org/`
[5] `http://jena.sourceforge.net/`
[6] `http://www.joseki.org/`
[7] `http://www.aktors.org/technologies/3store/`
[8] `http://www.franz.com/agraph/allegrograph/`

durch eine bereits betrachtete URI gegeben ist. Diese Nachteile lassen sich jedoch durch die Verwendung einer geeigneten Indexstruktur zumindest z.T. kompensieren. Dadurch lassen sich die besuchten Quellen dann auch über die Auswertung einzelner Queries hinaus zwischenspeichern und wiederverwenden. Eine solche Indexstruktur wird in [HHK+10] vorgeschlagen und eingehend untersucht. Zudem schlagen [LT10] explizit einen hybriden Ansatz vor, in dem der explorative Auswertungsansatz mit dem einer informierten, möglicherweise verteilten, Anfragenauswertung kombiniert wird.

Ein dritter Ansatz, der auch im Weiteren der Arbeit betrachtet wird, ist die föderierte Abfrageverarbeitung innerhalb sogenannter föderierter Datenbanksysteme. Eine föderierte Datenbankarchitektur umfasst ein zentrales Datenbanksystem, das selbst, abgesehen von zentralen Systemdaten, keine eigenen Datenbestände verwaltet. Stattedessen findet Föderation, eine virtuelle Form der Datenintegration, statt. Dazu sind ihm mehrere Datenbanken unterstellt, die ihren jeweils eigenen Datenbestand für Abfragen seitens des übergeordneten Systems einbringen. Diese behalten währenddessen einen gewissen Grad an Autonomie bei und sind mehr oder weniger lose an das Gesamtsystem gekoppelt. Der Zugriff lokaler Anwendungen auf einzelne Teilnehmer-DBS bleibt dadurch von der Föderation unbehelligt. Eine zentrale Komponente, der Föderierungsdienst oder Federator, sorgt für diese Föderation, also für die Verbindung der Teilnehmerdatenbanken. Dabei wird globalen Anwendungen eine globale Sicht, ein globales Schema, bereitgestellt und ein einheitlicher Zugriff ermöglicht.

Der Föderationsansatz, wie er in klassischen Datenbanksystemen zur Anwendung kommt, ist besonders auch für Linked Data geeignet. Die Sicht, die dem Föderationskonzept vorausgeht, ist verträglich mit der tatsächlichen Datenverteilung von Linked Data einerseits und den Linked-Data-Prinzipien andererseits. Daraus ergeben sich auch einige Vorteile bei der Anfragenverarbeitung, wie dies in [HMZ10] diskutiert wird. Zum einen umgeht man, im Unterschied zum Datawarehousing, die ansonsten auftretende Datenredundanz. Viel entscheidender ist jedoch die Möglichkeit der parallelen Beantwortung einzelner (Teil-)Queries, die sich aus der Autonomie der Föderationsteilnehmer ergibt. In demselben Zusammenhang sind Vorteile zu sehen, die sich dadurch ergeben, dass etwaige Ressourcenengpässe (engl. *bottlenecks*) bei der Ergebniserzeugung auf Teilnehmerdatenbanken lokalisiert sind und sich dadurch parallelisierte Auswertungsvorgänge gegenseitig weniger wahrscheinlich blockieren. Zudem ergeben sich Vorteile bezüglich der Skalierbarkeit, zumal es günstiger ist einen großen Datenbestand der Datenballung nach und gemäß dem Prinzip der „Trennung der Belange“ separiert

zu verwalten. Aber auch jenseits der Systemperformance bietet Föderation günstige Eigenschaften, die einem Konsumenten des Semantic Web zu Gute kommen. Dies betrifft die Autorität und Provenienz bezüglich veröffentlichter Daten. Seit jeher ist es ein Ziel des Semantic Web auch die Herkunft und Vertrauenswürdigkeit bestimmter Daten nachvollziehen zu können. Dies wird durch das Anlegen von Kopien z.B. beim Datawarehousing erschwert. Verbleiben Daten jedoch in demselben System, in dem sie erzeugt werden, wie dies durch Föderation prinzipiell möglich ist, erübrigt sich dies und die Vertrauenswürdigkeit bestimmter Daten reduziert sich auf die Autorität, die einer Datenquelle allgemein zugesprochen wird.

3 Related Work

Das Problem fehlerhafter Schätzungen bei der Query-Optimierung, das auch Gegenstand der vorliegenden Arbeit ist, wird in der Datenbankforschung bereits seit langer Zeit explizit aufgegriffen. Dabei zeigt sich, dass es verschiedene Gründe gibt, die eine Beobachtung und Behandlung (z.B.) auftretender Schätzfehler im Laufe der Query-Optimierung motivieren. Während anfänglich im Rahmen von Systemanalysen die manuelle Validierung neuer Optimizer im Vordergrund stand, wurde und wird dadurch in neueren Entwicklungen zunehmend der gesteigerten Dynamik moderner Informationssysteme Rechnung getragen. Im Folgenden sollen zur thematischen Einordnung konkrete Arbeiten vorgestellt werden, die (zumindest indirekt) eine Beobachtung der Performance kostenbasierter Query-Optimizer zum Thema haben.

3.1 Statische DBMS-Validierung

Ein Aspekt der Datenbankforschung, der in der Literatur bereits vergleichsweise früh behandelt wird, ist die statische, also einmalige manuelle, Validierung bestehender kostenbasierter Optimizer. Chronologisch ist die Separierung von Datenmodell von der physischen Datenhaushaltung durch die Einführung des Relationalen Schemas durch Codd [Cod70] als ursächlich für das Aufkommen gesonderter Query-Optimizer zu betrachten. Angefangen mit dem DBMS System R [SAC$^+$79] setzte in der Folge das Aufkommen einer Fülle kommerzieller und freier Systeme ein, deren Optimizer-Performance erst zu untersuchen bzw. unter Beweis zu stellen war. Als Beispiel seien hier drei Arbeiten näher betrachtet, in denen eine solche Validierung diskutiert wird.

So wird in [ML86b] das DBMS R*, eine Weiterentwicklung von System R, auf die Güte der Kostenschätzung bei lokaler Query-Auswertung untersucht. Diese ist systemabhängig und wird daher unter verschiedenen Rahmenbedingungen vergleichend untersucht. Zu den Fragestellungen, die im Rahmen der jeweiligen Validierungen aufgegriffen werden, zählen (siehe [ML86b]):

- Welche Systemparameter haben einen besonderen Einfluss auf den Optimizer und dessen Performance?
- In welchen Bereichen des Raums über diesen Parametern verwirft der Optimizer den besten Plan oder wählt gar den schlechtesten?

- Werden diese Systemparameter überhaupt „genau“ abgebildet?
- Kann das Kostenmodell vereinfacht werden, um eine schnellere Optimierung zu erhalten?
- Welche Statistiken über dem Datenbestand unterstützen den Optimizer am besten?

Auch werden im Anschluss, aufbauend auf den Befunden, mögliche Verbesserungen vorgeschlagen. Die Validierung orientiert sich vor Allem an der Kostenberechnung bzw. der Interpretation dieser durch den Optimizer. Die Kosten, die durch den Optimizer von R* minimiert werden, berechnen sich aus einer Funktion, in der je (Teil-)Query eine gewichtete Summe aus vier Komponenten gebildet wird. Diese sind: die Anzahl der nötigen Instruktionen (`#__instr`), die Anzahl der Lese-/Schreibzugriffe(`#__i/os`), die Anzahl von Netzwerknachrichten(`#__msgs`) und die Zahl der dabei übertragenen Bytes (`#__bytes`). Jede dieser Größen geht nach einem systemspezifischen Gewicht in die Summe ein. Die Gewichte sind so dimensioniert, dass eine Konvertierung in Millisekunden stattfindet.

$$\begin{aligned} \texttt{R*\_\_cost}[ms] &= W_{CPU} * (\texttt{\#\_\_instr}) + W_{I/Os} * (\texttt{\#\_\_i/os}) \\ &\quad + W_{msgs} * (\texttt{\#\_\_msgs}) + W_{bytes} * (\texttt{\#\_\_bytes}) \end{aligned}$$

Ausgangspunkt für die Validierung bietet die Vorbereitung für eine umfängliche Erhebung geeigneter Messdaten. Dazu schlagen Mackert und Lohman in [ML86b] einen Mechanismus vor, der es ihnen erlaubt je Query anfallende Schätzungen und Leistungsdaten gezielt zu messen und schließlich abzuspeichern. Bei den Messdaten handelt es sich im Wesentlichen um Zählwerte aus bestimmten Systemvariablen, die wie im Vorgängersystem System R, automatisch erzeugt werden. Insbesondere die geschätzten Kosten, die bei der Auswertung festzustellenden tatsächlichen Kosten sowie die Rahmenbedingungen der umgebenden Ausführungsumgebung und die Datensituation (Werteverteilungen) sind dabei von Interesse. Um für die Datenanalyse aus der steuernden Testanwendung heraus auf einzelne dieser Messungen zugreifen zu können, müssen diese zunächst ausgelesen werden. Neben den normalen Benutzertabellen, werden dafür dedizierte Tabellen angelegt, in denen die bei der Testausführung anfallenden Schätzungen und Leistungsdaten eingepflegt werden können. Um den Testverlauf steuern zu können und Messungen schließlich auch in die Tabellen zu schreiben erweitern Mackert, Lohman den Befehlssatz der SQL-Implementation in R* um drei weitere spezielle Befehle.

Mit `EXPLAIN` kann das DBMS von einer Anwendung aus dazu angewiesen werden, die bei der Planung und Optimierung einer Query anfallenden Daten

in die vorbereiteten Tabellen zu schreiben. Das betrifft hier insbesondere die Kostenschätzungen für die Query-Ausführung. Durch die Angabe des Statements `COLLECT COUNTERS` werden die aktuellen Systemzähler ausgelesen und ebenfalls mit Zeitstempel versehen in die vorgesehenen Datentabellen übernommen. Dies betrifft Vermerke über den Ressourcenverbrauch, etwa die Zähler für die bis zum jeweiligen Zeitpunkt ausgeführten CPU-Instruktionen, Lese-/Schreib- und Pufferzugriffe.

Schließlich lässt sich der Optimizer in der angepassten Testversion von R* durch Angabe des neuen Statements `FORCE OPTMIZER` dazu bringen, einen mit einer eindeutigen Nummer bezeichneten Plan an den Query-Prozessor auszugeben, unabhängig davon, ob dieser auch als optimal eingestuft wurde.

Aufbauend auf dieser Ausstattung zeigen die Autoren, wie der eigentliche Validierungsvorgang gesteuert wird. Eigens zu diesem Zweck entwickelte Anwendungen greifen auf das präparierte Test-DBMS zu und beeinflussen den Testablauf. Zum einen wird eine Menge verschiedener Queries an dem Datenbanksystem ausgewertet, die dabei anfallenden Daten über die angesprochenen Befehle in die erzeugten Tabellen geschrieben. Zum anderen werden über weitere Anwendungen die im Laufe des Test festgeschriebenen Daten aus den angesprochenen Tabellen ausgelesen, geeignet aufbereitet und ausgedruckt bzw. grafisch ausgegeben. Die Menge der Queries wird während des Tests mehrfach ausgewertet, der Datenbestand jeweils nach zuvor festgelegter Verteilung und Umfang variiert und Puffer des DBMS zurückgesetzt.

Bezüglich der eigentlichen Validierung, deren Ergebnisse in [ML86b] diskutiert werden, sind noch mehrere interessante Aspekt anzumerken, zumal sie die Motivation einer separaten Optimizer-Validierung exemplarisch vor Augen führen. Es kann ein allgemein signifikanter Kosteneinfluss seitens der CPU-Komponente festgestellt werden. Gleichzeitig wird gezeigt, dass Zugriffe (z.B. Prädikatauswertungen) direkt auf Ebene des Storage-Systems die CPU weniger in Anspruch nehmen, als auf Ebene der darüber liegenden ausführenden DBMS-Komponente. Da im Allgemeinen jedoch beide Situationen vorkommen, kann eine Präzisierung des Kostenmodells bezüglich der CPU-Kostenberechnung motiviert werden. Durch den Vergleich der während der Testdurchführung beobachteten Kostenschätzungen und der tatsächlich festgestellten Kosten kann zudem die Optimierung bei Joins genau untersucht werden. Es gelingt den Autoren festzustellen, dass der Optimizer für Joins im Allgemeinen tatsächlich den besten Plan liefert. Als ein Sonderfall wird die Situation beobachtet, bei der ein Tabellenindex parallel zum Laden der Tabellenseiten über den DBMS-Puffer abgewickelt wird. Die Kosten hierfür werden unterschätzt, ggf. gar der schlechteste Plan bevorzugt. Als Verbesse-

rung wird hier eine Verfeinerung des Kostenmodells vorgeschlagen, durch die die Nebenläufigkeit beider Prozesse einbezogen wird. Schließlich kann gezeigt werden, dass die einfache Schätzung für Join-Kardinalitäten die Wahl von Nested-Loop-Joins ungerechtfertigt benachteiligt. Dadurch wird eine zusätzliche Verfeinerung der Schätzverfahren durch eine Unterstützung mit Hilfe statistischer Daten motiviert.

Neben dieser Arbeit, die für die vorliegende Arbeit exemplarisch betrachtet wird, sind in der Literatur weitere Arbeiten zu ähnlich gelagerten Problemstellungen zu finden. Von denselben Autoren wird in [ML86a] eine gesonderte Validierung der verteilten Query-Verarbeitung in R* unternommen. Die Problemstellung dabei ist komplexer, da sich, im Unterschied zur obigen Situation, aufgrund der verteilten Queries nun auch Aspekte der Netzwerkauslastung in den Kosten einer Query-Auswertung niederschlagen. Das betrifft die Komponenten (`R__#msgs`) und (`R__#bytes`) der obigen Kostenfunktion. Die Autoren beschreiben die Validierung mit einer Testkonfiguration, die, bis auf Anpassungen aufgrund der Verteilung des Datenbestands, derjenigen in [ML86b] entspricht. Es kann im Rahmen dieser Validierung gezeigt werden, dass die Kostenschätzung der Message-Komponente bei bekannten Tabellenkardinalitäten sehr genau ist. Schätzfehler bei Kosten treten hier jedoch dann auch durch etwaige unzureichende Kardinalitätsschätzungen bei Joins auf. Dadurch wird der Vorschlag aus [ML86b] für eine verfeinerte Kardinalitätsschätzung für den R*-Optimizer weiter bekräftigt. Zudem kann gezeigt werden, dass das Kostenmodell bei verteilten Queries nicht auf die Modellierung der lokalen Aspekte des CPU- und Lese-/Schreibzugriffs verzichten kann, da diese auch hier einen bedeutsamen Einfluss auf die Gesamtkosten haben.

In [ML89] wird schließlich ein präzisiertes Kostenmodell vorgeschlagen, dass die Kostenschätzung von Index-Scans bei Annahme eines begrenzten DBMS-Puffers mit LRU-Strategie zulässt. Die Plausibilität der entworfenen Modellierung wird ebenfalls durch eine empirische Validierung untersucht. Das Testsystem ist auch hier dasselbe wie oben. Bei dem beschriebenen Testablauf werden wiederum verschiedene Parameter der Daten- und Systemkonfiguration variiert. Es gelingt zu zeigen, dass das vorgeschlagene Modell sowohl bezüglich der geschätzten Zahl der Seitenzugriffe, als auch nach Betrachtung des dazugehörigen relativen Schätzfehlers im Allgemeinen bessere Ergebnisse liefert, als das ursprüngliche von System R übernommene Kostenmodell.

3.2 Anpassbare Kostenmodelle

Einen anderen Blickwinkel auf die Performance gegebener Query-Prozessoren wird in Systemen eingenommen, in denen eine Kalibrierung (z.B.) des Kostenmodells auf Basis permanenter oder periodischer Leistungsmessungen unternommen wird. Um auch in einer dynamischen Anwendungsumgebung den Query-Optimizer eines (z.B. föderierten) DBMS adäquat zu unterstützen, werden entsprechend ausgestattete Kostenmodelle mit der Zeit automatisch angepasst.

Du et al. präsentieren in [DKS92] eine solche Herangehensweise. Um in einem heterogenen DBMS von zentraler Stelle aus auch verteilte, d.h. von mehreren beteiligten Systemen anfragende Queries gut optimieren zu können, müssen die teilnehmenden DBMS das Zentralsystem an ihren eigenen Kostenmodellen teilhaben lassen. Bei genau darauf ausgerichteten proprietären Systemen ist dies denkbar. Für den allgemeinen Fall jedoch schlagen die Autoren ein generisches Kostenmodell vor, das typische physische Operatoren für Selektion und Join von Tupeln abschätzen kann. Für die Kostenfunktionen sind jeweils mindestens drei Koeffizienten zu bestimmen, die die Kosten der Zugriffsinitialisierung, des Auffindens und Verarbeitens einzelner Tupel angeben. Um diese Koeffizienten für ein konkretes DBMS mit der notwendigen Vorhersagbarkeit für die verschiedenen Operatoren zu bestimmen, zeigen die Autoren die Erzeugung synthetischer, beliebig skalierbarer Kalibrierungstabellen. Diese beinhalten mehrere Attribute mit exakt vorherbestimmten Eigenschaften bezüglich Sortierung und Wertverteilung. Bei der eigentlichen Kalibrierung werden speziell erzeugte Queries an den künstlichen Daten ausgewertet. Mit den Ergebniskardinalitäten, die mit Erzeugung der Daten bereits bekannt sind, und den zu beobachtenden Kosten der Auswertung lassen sich die Koeffizienten dann abschätzen. Das auf diese Weise initialisierte generische Kostenmodell orientiert sich an der Leistungsfähigkeit des Systems, wie sie sich aus Benutzersicht darstellt. Das DBMS selbst wird dabei weitestgehend als Blackbox behandelt. Die Autoren beschreiben die Kalibrierung dreier konkreter Systeme: DB2, Allbase und Informix. Im Laufe einer Gegenüberstellung der geschätzten und der tatsächlichen Kosten, kann gezeigt werden, dass das generische Kostenmodell akzeptable Schätzungen erlaubt, deren Abweichung gegenüber den tatsächlichen Kosten zumeist weniger als 20% betragen.

Unter Bezugnahme auf [DKS92] beschreiben Gardarin, Sha und Tang in [GST96] ein objektorientiertes föderiertes Datenbanksystem. In diesem werden mehrere verteilte Datenquellen über gemeinsame ODMG-Schnittstellen

zu einem Datenbanksystem zusammengefasst. Wie in föderierten Datenbanksystemen allgemein, ergibt sich auch hier ein Problem. Für eine gute Optimierung verteilter Queries über die an der Föderation beteiligten DBMS sind Informationen über diese DBMS vonnöten, die ohne weiteres zunächst jedoch nicht vorliegen oder zumindest nicht proaktiv ausgetauscht werden. Um dem zu begegnen, schlagen die Autoren hier eine Optimizer-Architektur vor, in der lokale Kalibrierungen, wie in [DKS92] vorgeschlagen, unterstützt bzw. erfordert werden. Der Optimizer des übergeordneten Föderationssystems besitzt dazu ein Kostenmodell das mit einem Datenkatalog in Verbindung steht. Über diesen Datenkatalog können bei der Optimierung und Kostenschätzung verteilter Queries schematische und kostenrelevante Informationen eingeholt werden. Sämtliche Teilnehmer-DBMS müssen eine Schnittstelle der Optimizer-Architektur implementieren, über die, nach der in [DKS92] skizzierten Idee, Koeffizienten eines generischen Kostenmodells abgeschätzt und abgespeichert werden. Um damit dann den Datenkatalog zu füllen, werden vom Betreiber des föderierten Systems Anfragen über die lokal erhobenen Kalibrierungsdaten ausgewertet. Auch in [GST96]kann schließlich in Testdurchläufen gezeigt werden, dass die Kosten der ausgewerteten Queries in annehmbarer Weise abgeschätzt werden.

Dasselbe Kostenmodell findet auch in [NGT98] Verwendung. Beschrieben wird eine sogenannte Mediator-Architektur für die Zusammenführung verschiedener heterogener und verteilter Datenquellen. Diese werden dazu über Wrapper einheitlich aufgegriffen und bereitgestellt. Diese Wrapper können Informationen, die über die jeweiligen Datenquellen erhoben wurden, proaktiv an das Mediatorsystem weitergeben. Dazu zählen neben Statistiken über den Datenbestand (Kardinalitäten, Item-Counts, min-/max-Werte für Attribute, etc.) auch spezifische Kostenformeln, die nur aus Sicht des Wrappers bekannt sind. Diese werden an eine ebenfalls bekanntgegebene Interface-Definition des Wrappers angehangen. Dadurch kann das globale generische Kostenmodell des Mediators präzisiert und eine quellenspezifische und damit genauere Kostenschätzung ermöglicht werden.

Die Idee einer zentral verwalteten Datenbank für kostenrelevante Informationen wird bereits durch Adali et al. in [ACPS96] aufgegriffen. Die Autoren beschreiben Optimierungsstrategien in einem verteilen Mediatorsystem. Der Mediator verarbeitet regelbasierte Queries und setzt Teil-Queries in Form von Procedure-Calls an Programme in verteilten Ausführungsumgebungen um, dabei errechnete Ergebnisse werden als Anfragenergebnis zurückgegeben. Der Optimizer wählt den günstigsten Plan, dies sind hier Folgen von Procedure-Calls, für eine Query nach tatsächlichen Kosten bereits in der

Vergangenheit ausgewerteter (Teil-)Queries. In einer dedizierten Kostendatenbank werden bei der Auswertung einzelner Call-Instanzen erzeugte Kostenvektoren vorgehalten. Ein solcher Kostenvektor beinhaltet dabei die Call-Instanz, die Zeitpunkte zu denen das erste und dann alle Ergebniselemente gelieferten wurden und die Ergebniskardinalität. Um einer möglichen Speicherplatzknappheit aufgrund zu vieler dieser Daten vorzubeugen, betrachten die Autoren zusätzlich eine Komprimierung durch verlustlose und verlustbehaftete Summary-Verfahren, die die Kostenstatistiken für einzelne Datenquellen zusammenfassen. Bei der Kostenschätzung neuer Procedure-Calls werden dann die gemittelten Zeiten und Kardinalitäten der zurückliegenden Instanzen derselben Calls einbezogen.

Der Einsatz eines ähnlichen Kostenrepositoriums wird auch von Hidalgo et al. in [HPlG06] präsentiert. Das beschriebene Einsatzszenario ist hier die Mediation verschiedener verteilter webbasierter Datenquellen. Dies sind hauptsächlich HTML-Dokumente, die über Wrapper bereitgestellt werden.

Die bisher geschilderten Arbeiten behandeln die kostenbasierte Optimierung in verteilten Systemen, bei denen über lokale Kosteneinflüsse keine genauen Informationen weitergegeben werden und die stattdessen generisch behandelt werden. Ein anderer Gesichtspunkt in diesem Zusammenhang spielt die Dynamik solcher Systeme. Wie in [RZL04] dargelegt wird, lassen sich kostenrelevante Veränderungen beteiligter Teilsysteme nach der Häufigkeit ihrer Veränderung in drei Klassen einteilen.

Die Autoren unterschieden demnach (siehe [RZL04]):

- Typ 1: Kosteneinflüsse mit hoher Änderungsfrequenz: CPU-Last, Anzahl der I/O-/Netzwerk-Zugriffe pro Sekunde Speicherauslastung, etc.
- Typ 2: Kosteneinflüsse, die sich gegenüber Typ 1 langsam verändern, bezogen auf die Dauer einer Query-Auswertung nahezu unverändert bleiben: Zugriffsimplementation, physische Datenverteilung, schematische Konfigurationen seitens lokaler DBMS
- Typ 3: Kosteneinflüsse die sich vernachlässigbar selten ändern (z.B. Hardwarekonfiguration lokaler DBMS).

Eine generische Kostenfunktion y lässt sich nach [ZL94] als Skalarprodukt aus einem Koeffizientenvektor $\vec{\beta}$ und einem Vektor von explizit modellierten Kostenfaktoren $\vec{x}$ (Kardinalitäten von Basistabellen und (Teil-)Ergebnissen, Tupelgrößen, physischen Eigenschaften der Datenbasis, etc.) auffassen. Im Unterschied zu dem Ansatz aus [DKS92], zeigen Zhu und Larson einen multiplen Regressionsansatz für die Koeffizientenschätzung mit Hilfe von Query-Sampling. Dabei werden Gruppen „ähnlicher“ Queries (z.B. nach ihrer Struk-

tur) betrachtet. Für die Auswertungen von Queries einer Gruppe fallen Beobachtungen von Kosteneinflüssen $\vec{x}_i$ und der tatsächlichen Kosten y_i an. Durch Least-Square-Fitting über (y_i,$\vec{x}_i$) lässt sich der Koeffizientenvektor $\vec{\beta}$ abschätzen. Das Kostenmodell lässt sich über die darüber abgeschätzten Kostenkoeffizieten dadurch immer weiter anpassen. Mit zunehmender Zahl solcher Beobachtungen ist eine Berechnung der Koeffizienten jedoch teuer und möglicherweise problematisch im Speicherverbrauch.

Rahal et al. zeigen in [RZL04] ein evolvierendes Kostenmodell, mit dem sich langsame Veränderung (Typ 2) verteilten System kompensieren und Kostenschätzung dadurch in einem angemessen Maß halten lassen. Auch hier wird wie in [ZL94] Query-Sampling auf Basis eines Regressionsmodell unternommen. Jedoch präsentieren die Autoren zwei effiziente rekursive Berechnungsansätze, bei denen für ein aktuelles Kostenmodell nur die letzten k Query-Samples Eingang in die Koeffizientenabschätzung finden. Bei der Beobachtung neuer Queries, wird auf Basis des alten Kostenmodells ein neues Kostenmodell berechnet, indem der Einfluss der ältesten Query heraus- und der Einfluss der neuesten Query eingerechnet wird. Die Komplexität der Berechnung lässt sich dann durch die Wahl der Samplinggröße k variieren und entsprechend mit der Genauigkeit der Modellschätzung abwägen.

Kostenschätzungen in DBMS basieren im Kern auf Statistiken über den Datentabellen. Ebenso können Fehler in der Kostenschätzung auf Fehler in der Statistiksammlung des Kostenmodells zurückzuführen sein. In [SLMK01] diskutieren Stillger et al. eine Anpassung des Optimizers des DB2-DBMS, die auf fehlerhafte Statistiken kompensatorisch reagieren kann – LEarning Optimizer (LEO). Zu der Architektur des Optimizers werden dazu vier weitere Komponenten hinzugefügt. Eine Komponente, die den Plan der optimalen Query speichert, eine Messkomponente, die sich in den Query-Prozessor einfügt, eine Analysekomponente und eine so bezeichnete „LEO Feedback Exploitation“-Komponente. Insgesamt, wird der Optimizer dadurch in die Lage versetzt aus begangenen und beobachteten Schätzfehlern zu lernen. Im Ablauf der Abfrageverarbeitung wird zunächst der Plan einer optimalen Query nach Ausgabe aus dem Optimizer zusammen mit den Kardinalitätsschätzungen der enthaltenen physischen Operatoren zwischengespeichert. Die Messkomponente zählt im Anschluss, während der Auswertung des Plans, für jeden der Operatoren die tatsächliche Ergebnisgröße. Damit lassen sich je Operator geschätzte (*est.*) und tatsächliche (*act.*) Kardinalität einander gegenüberstellen. Um Schätzfehler bei der Selektivitäts- und Kardinalitätsschätzung ausgleichen zu können, verwaltet LEO eine Datenbank für Ausgleichsfaktoren. Diese Faktoren sollen die ursprüngliche Schätzung des DB2-

Optimizers multiplikativ ergänzen und dadurch korrigieren – die ursprünglichen (möglicherweise fehlerhaften) Statistiken bleiben jedoch weiterhin erhalten. Die Datenbank besteht aus Tabellen, in denen aktuelle Korrekturwerte für die Kardinalitäten der Basistabellen und die Selektivitäten von instanziierten Selektions- und Join-Prädikaten aufbewahrt werden. Die Analysekomponente von LEO beobachtet je Operator den relativen Fehler der Selektivitäts-/Kardinalitätsschätzung (hier $|est. - act.|/act.$) und errechnet im Falle eines relativen Fehlers über 0.05 einen aktuellen Ausgleichsfaktor bezüglich *act.* - etwaige alte Faktoren werden dabei heraus gerechnet. Für die Abschätzung neuer Queries während der Optimierung, können dann über die Exploitation-Komponente passende Faktoren aus der Datenbank abgerufen und eingerechnet werden.

Aufbauend auf [SLMK01] präsentieren Ewen, Ortega-Bindenberger und Markl eine Ausweitung von LEO auf den Fall föderierter DBMS [EKMR06a]. Neben der Anpassung auf das Föderationsszenario, diskutieren die Autoren weitergehende Ergänzungen wie eine Korrelationsanalyse für abhängige Attribute. In Tests können die Autoren zeigen, dass die Ausführungszeiten der weitaus meisten getesteten Queries durch den Einsatz ihrer Lösung (z.T. um Größenordnungen) verbessert werden können.

3.3 Adaptive Query Processing

Ein gänzlich neues Paradigma für die Anfragenverarbeitung insgesamt ist das sogenannte Adaptive Query Processing (AQP). Diese Entwicklung wird durch das Aufkommen neuerer komplexer und dynamischer Informationssysteme befeuert. Beispiele sind Informationssysteme, in denen z.B. komplexe und langlaufende Abfragen, etwa zur Entscheidungshilfe in (geschäfts-)strategischen Fragestellungen, an einer Vielzahl verteilter Datenquellen ausgewertet werden. Angesiedelt sind solche Informationssysteme im Überlappungsbereich von Data Mining und Online Analytical Processing (OLAP) (siehe [DIR07]). Durch AQP wird mit dem konventionellen Phasenablauf des Query Processings(siehe Abschn. 2.4) gebrochen [DIR07, S. 3]. Statt der klaren Abfolge Parsen – Optimieren – Ausführen, die eine Query im Laufe ihrer Verarbeitung in klassischen DBMS durchläuft, wird die Ausführung bei AQP ggf. nach Eintreten bestimmter Ereignisse zugunsten einer Re-Optimierung von Teilen der Query unterbrochen. Dazu wird der Query-Prozessor zusammen mit dem Optimizer derart ausgestattet, dass es zu einer „Rückkopplung“ zwischen beiden Komponenten kommen kann, in deren Ver-

lauf sowohl die aktuell zu verarbeitende Query, als auch das Kostenmodell insgesamt „verbessert“ werden können (vgl. Abb. 3.1). Die oben beschriebene LEO-Ansatz schlägt bereits in diese Kerbe, ist jedoch weniger dynamisch. Der Query-Prozessor betreibt gewissermaßen permanent eine automatische selbstreferenzielle Validierung. Die Beobachtung (beispielsweise) der Schätzfehler, die im Laufe Betriebs auftreten, kann als Indikator für mangelnde Optimizer-Performance und damit (zumindest indirekt) als Anstoßpunkt für Re-Optimierungen dienen.

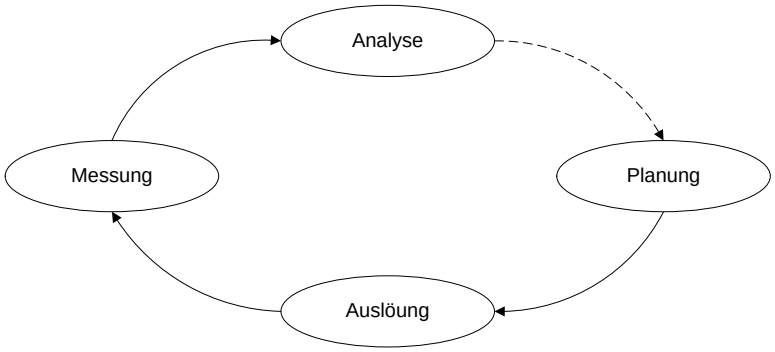


Abbildung 3.1: Grobskizze des Datenflusses bei Intra-Query AQP (entnommen aus [DIR07, S. 36]). Im Falle von Inter-Query-Adaption fehlte der durch Strichelung angedeutete Datenfluss, der Übergang zur Re-Optimierung.

Ein Beispiel für AQP wird von Markl et al. mit Progressive Query Optimization (POP) [MRS+04] präsentiert. Mit POP können noch während der Auswertung komplexer Queries (also mid-query) Abweichungen der Kardinalität des noch einlaufenden Ergebnisses von dem zugehörigen Schätzwert festgestellt werden. Sind diese signifikant, so wird eine Optimierung der (Teil-)Query wiederholt angestoßen. Zu POP gehört ein spezieller physischer Operator `CHECK`, der im Operatorbaum an mehreren Stellen, über den Wurzeln beliebiger Teilpläne, eingesetzt werden kann. Dieser Operator misst die Zahl der Ergebnisreihen, die über seine Eingabekante „fließen“, d.h. von dem unter ihm liegenden Teilplan ausgegeben werden (Input-Kardinalität). Jede Instanz eines `CHECK`-Operators hat gleichzeit als Argument ein vorberechnetes Validitätsintervall, in dem die tatsächliche Input-Kardinalität gemäß Einschätzung des Optimizers letztlich liegen muss. Liegt die tatsächliche Input-Kardinalität eines `CHECK`-Operators während der noch fortlaufenden Auswertung der Query außerhalb von $[l, u]$, so weist der betroffene POP eine Re-Optimierung an. Die tatsächliche Kardinalität a wird dann in die neue Optimierung mit einbezogen. Die Intervalle werden bei der Erzeugung der QEP (z.B.) bei Dynamic

Programming berechnet und orientieren sich an der Kostenschätzung alternativer (Teil-)Plänen, die bei der Optimierung zunächst verdrängt werden, und sind vergleichsweise konservativ gewählt. Unterstellt man, dass der aktuell verfolgte (Teil-)Plan optimal ist, müssten etwaige Abweichungen noch innerhalb der jeweiligen Intervalle liegen. Ist dies nicht der Fall, so wird durch die folgende Re-Optimierung auf einen neuen „optimalen“ Plan ausgewichen. Möglicherweise bereits materialisierte Teilergebnisse können dabei wiederverwendet werden, sofern dies im Rahmen der Re-Optimierung als vorteilhaft eingeschätzt wurde. Dazu können, neben den `CHECK`s, auch sogenannte Materialisierungspunkte innerhalb eines Plans festgelegt werden, an denen das eintreffende Teilergebnis zwischengespeichert wird. Mit Hilfe von POP können, wie in [MRS+04] demonstriert, in Einzelfällen Queries erzeugt werden, die im Vergleich zur normalen Abfrageverarbeitung ohne POP um bis zu zwei Größenordnungen besseren Antwortzeiten haben.

Anwendung finden die Verfahren von POP auch in [EKMR06b]. Darin stellen Ewen et al. eine Modifikation von POP für den Fall föderierter Queries vor. In der präsentierten Fallstudie, in der seitens der verteilten Datenquellen wenig statistische Information vorliegt, kann durch den Einsatz von POP eine allgemeine Verbesserung der Antwortzeiten gezeigt werden.

Ein früher Ansatz für AQP, der dem von POP ähnlich ist, findet sich bereits in [KD98]. Darin zeigen Kabra und DeWitt Dynamic Re-Optimization, ein Verfahren für die dynamische, mid-query Re-Optimierung eines Plans. Nach dem hier dargestellten Vorgang wird ein herkömmlich erzeugter Plan durch die bei der Kostenschätzung der einzelnen Operatoren angefallenen Informationen (z.B. Kardinalitätsschätzungen) annotiert. Ähnlich zu den `CHECK`-Operatoren aus [MRS+04], wird ein Plan an „wichtigen“ Stellen um `STATISTICS-COLLECTOR`-Operatoren ergänzt, die während des Pipelinings der Teilergebnisse Statistiken über diesen sammeln. Zu diesen Statistiken zählen z.B. die Kardinalität der Teilergebnisses, dessen durchschnittliche Tupelgröße sowie die Verteilung von Attributwerten. Diese können gegenüber den annotierten Schätzungen aus der Planungsphase eine Re-Optimierung begründen. Kabra und DeWitt betrachten jedoch auch eine aktualisierte Zuweisung von Ressourcen (z.B. die Speicherallokation) für die weitere Auswertung des aktuellen QEP auf Basis der neueren Statistiken, die eine Beschleunigung der Auswertung nach sich ziehen kann. Sollte jedoch nach bestimmten Heuristiken festgestellt werden, dass eine Re-Optimierung auf Grundlage der neuen Statistiken gegenüber der Fortführung der QEP (auch mit neuer Ressourcenzuweisung) eine kürzere Auswertungszeit zur Folge hätte, so wird diese angestoßen. Dies betrifft jedoch nur die Operatoren, deren Auswertung noch

nicht begonnen hat. Die Ergebnisse der bereits (z.T.) ausgewerteten Operatoren werden vollständig materialisiert und für die Auswertung des erneut optimierten Teilplans einbezogen. Kabra und DeWitt können in [KD98] experimentell eine Verbesserung der Antwortzeiten durch Re-Optimierung bei Queries mit 2 oder 3 Joins um bis zu 5% und bei Queries mit 4 oder Joins um bis zu 30% nachweisen – jeweils bei gleichverteilten (Zipf-verteilt[1] mit $z = 0$)Attributwerten. Sind die Wert nicht gleichverteilt (betrachtet werden zusätzlich $z = 0.3$ und $z = 0.6$), tritt dieser Effekt sogar etwas stärker zutage.

Einen anderen Beitrag zur Entwicklung im Bereich AQP leisten zwei Jahre nach [KD98] Avnur und Hellerstein. Sie diskutieren in [AH00] einen neuen speziellen physischen Operator, Eddy genannt. Die Idee hinter Eddies ist anders als bei der mid-query Re-Optimization von Plänen nach [KD98] oder [MRS+04]. Eine konkrete Eddy-Instanz repräsentiert einen vollständigen QEP. Als Eingabe bezieht ein Eddy-Operator Tupel der Basisrelationen, über die die Abfrage ausgewertet werden soll. Zusätzlich sind sämtliche Operator-Instanzen, aus denen ein herkömmlicher äquivalenter QEP bestünde, ohne Ordnungspräferenz an den Eddy gebunden. Der Eddy-Operator gibt zwischenzeitlich gebildete Tupel der entstehenden (Teil-)Ergebnisse (zu Beginn aus den Basisrelationen) während der Auswertung an diejenigen Operatoren aus, die mit diesen "kompatibel" (z.B. macht es keinen Sinn ein Tupel aus Relation R an $\sigma_p(T)$ zu übergeben) sind und die von dem Tupel bisher nicht passiert wurden. Die von diesen Planoperatoren ausgegebenen und manipulierten Tupel werden sofort an den Eddy zurückgegeben. Hat ein Tupel schließlich alle angeschlossenen Operatoren passiert, handelt es sich also um ein Tupel des Endergebnisses, wird es schließlich auch vom Eddy ausgegeben. Die Reihenfolge oder Route, mit der ein bestimmtes Tupel nun die einzelnen Planoperatoren durchläuft, kann je Tupel neu festgelegt werden und lässt sich damit feingranular kostenoptimierend anpassen. Als Entscheidungsgrundlage dienen auch hier Statistiken, die während der Ausführung des QEP gesammelt werden. Die Planausführung, die damit zustande kommt, entspricht effektiv dem kontinuierlichen Wechsel alternativer Pläne nach dem obigen Re-Optimierungsansatz.

[1] Frequenz f_z von Wert v aus Domäne $D \subset \mathbb{N}$ ist $f_z(v) = \frac{1/v^z}{\sum_{i=1}^{max_D} 1/i^z}$

4 Problemstellung und Lösungskonzept

In föderierten Datenbanksystemen findet der virtuelle Zusammenschluss mehrerer entfernter, verteilter Datenquellen statt, der von einem zentralen System verhandelt wird (siehe Abschnitt 2.5). In konkreten hier betrachteten Systemen, wie z.B. den in [GST96, NGT98] dargestellten Entwicklungen, ist die dynamische Einbindung einzelner solcher Datenquellen von vornherein vorgesehen und wird entsprechend unterstützt. Als Konsequenz werden die Datenquellen intern durch statistische Zusammenfassungen repräsentiert. Informationen dieser Art werden hier dem Kostenmodell des zentralen Optimizers eingegeben, sodass dieser daraufhin Abfragepläne in ihren zu erwartenden Kosten erst abschätzen kann. Eine Kostenschätzung selbst basiert auf Schätzungen der Kardinalitäten und Selektivitäten der in einem Plan beinhalteten Operatoren. Solche Schätzungen hängen von der Kenntnis aktueller Item-Counts (Anzahl verschiedener Items/Elemente des Datenbestands), Werteverteilungen verschiedener Attribute und Abhängigkeiten von Attributwerten ab, die im abzufragenden Datenbestand vorherrschen. Wie gesehen, ist dies ein Weg dem Umstand zu begegnen, dass die jeweiligen Datenquellen von sich aus im Allgemeinen keine aktuellen Statistiken vorhalten und diese auch auf Anfrage nicht an das Zentralsystem weitergeben. In den im Weiteren in Betracht gezogenen Föderationssystemen herrscht nur eine sehr lose Kopplung zu den Datenquellen vor. Der proaktive Austausch statistischer Hilfsinformationen für die Anpassung des zentralen Kostenmodells ist darin also ebenso wenig vorgesehen.

Leider gehen mit dem Ansatz eines zentral verwalteten Katalogs von Datenstatistiken verschiedene Probleme einher, wenn man unterstellt, dass die Bestände der Datenquellen veränderlich sind. Zum einen stellt eine Datenstatistik natürlich nur eine Momentaufnahme der betrachteten Daten dar. Die statistische Charakteristik der Daten kann in der Zwischenzeit mehr oder weniger stark fluktuieren. Bedeutsame Veränderungen der Daten werden aber erst wieder durch eine neuere Datenstatistik abgebildet. In dem Zeitraum, in dem eine Datenstatistik verwendet wird, könnte es somit zu einer Auswahl und Ausführung suboptimaler Pläne kommen, sollten gleichzeitig Änderungen am Datenbestand vorgenommen werden. Wenn dann auch die Auswahl der zu verwendenden Datenquellen für die Beantwortung von Queries auf

obsoleten Datenstatistiken beruht, könnte es zudem zu einer ungenügenden Zuteilung von Teil-Queries zu den Datenquellen kommen - die Teil-Queries, und damit die gesamte Query, würden dann nur unzureichend beantwortet. Zum anderen ist die vollständige Erhebung der Statistiken eines Datenbestands für die Erzeugung einer Datenstatistik im Allgemeinen sehr aufwändig. Es schließt sich also in der Praxis von vornherein aus eine periodische oder gar permanente Neuerhebung sämtlicher Datenstatistiken während des Betriebs des Datenbanksystems vorzunehmen.

In den folgenden Abschnitten werden drei Verfahren vorgestellt, mit Hilfe derer die Erkennung von veralteten bzw. obsoleten Datenstatistiken ermöglicht werden soll. Nach Erkennung und Meldung einer solchen Situation seitens des zentralen Datenbanksystems (genauer dem Federator), soll dadurch eine Neuerhebung oder -Anforderung der jeweiligen tatsächlichen Datenstatistik ausgelöst werden können. Entscheidend ist, dass dies dann vorgenommen wird, wenn es als notwendig erachtet wird - nicht unnötig früh oder zu spät, in jedem Fall aber begründet auf konkreten Beobachtungen. Als Entscheidungsgrundlage werden durch die einzelnen Verfahren hier Fehlerbeobachtungen herangezogen, die durch einen Query-Feedback-Ansatz für jede der betrachteten Datenquellen separat erzeugt und verfolgt werden. Insgesamt soll so zumindest indirekt eine Feedback-Schleife realisiert werden, wie sie im vorherigen Kapitel bereits beschrieben wurde (siehe auch Abbildung 3.1).

Die Herangehensweise, die in dieser Arbeit erarbeitet und dargelegt werden sollte, basiert nun auf einer stark abstrahierten Sicht auf das vorliegende Problem. Diese Sicht soll im Folgenden kurz dargestellt werden.

4.1 Problemabstraktion

Systemmodell

Für die folgende Diskussion sei ein föderiertes Datenbanksystem f abstrahiert durch

$$f = (Q, P, plan, M_{cost}, S_{fed}).$$

Dabei sind

- Q, die Menge aller durch f annehmbaren Query-Ausdrücke
- P, die Menge aller Pläne über der physischen Algebra von f. P_q bezeichnet im Folgenden die Menge aller mit einer Query q verträglichen

Pläne.

- *plan*, die eine Funktion, die für ein $q \in Q$ einen optimalen Plan $p \in P_q$ liefert
- $S_{fed} = \{src_i\}$, die Menge aller an der Föderation beteiligten Datenquellen
- $M_{cost} = (cost, C_{dstats}, statistic)$, mit einer Kostenfunktion $cost : P \rightarrow \mathbb{R}_{\geq 0}$, die einem gegebenen Plan die geschätzten Ausführungskosten zuweist, und einem Datenstatistikkatalog $C_{dstats} = \{dstat_{src_i}\}$ mit der zugehörigen bijektive Funktion $statistic : S_{fed} - > C_{dstats}$, die einer Datenquelle $src \in S_{fed}$ eine Datenstatistik $statistic(src) = dstat_{src}$ aus C_{dstats} zuweist.

Ein solches System f nimmt eine Query q aus Q entgegen, mit dem Ziel, diese möglichst schnell und vollständig zu beantworten. Das System implementiert eine Funktion *plan*, die eine Query auf einen optimalen Plan über der physischen Algebra des Systems abbildet. Abstrakt ist *plan* dann gegeben mit

$$plan(q) = \underset{p \in P_q}{argmin}\, cost(p).$$

Es wird also die Kostenfunktion *cost* des Kostenmodells minimiert und damit ein der Schätzung nach optimaler Ausführungsplan p_q der Query erzeugt. Die Kostenfunktion ist selbst also abhängig von Schätzungen, nämlich der Schätzung der Kardinalitäten all der Operatoren, die in dem abzuschätzenden Plan enthalten sind. Für die hier betrachteten Systeme kommt dabei auf folgende Weise der Datenstatistikkatalog C_{dstats} zum Einsatz.

Ein durch *plan* generierter Plan p bildet im Allgemeinen eine föderierte Query ab und setzt sich somit aus einer Menge separater Teilpläne zusammen. Im Laufe der Planung einer solchen Query werden für einzelne Teilpläne identifiziert, die bei der folgenden Auswertung jeweils separat an einer der Datenquellen in S_{fed} auszuwerten sind. Dazu werden anhand der in den Teilplänen aufgeführten Relationen und Attributen Datenquellen in C_{dstats} nachgeschlagen, die ebendiese selbst in ihrem Bestand aufweisen. Die Teilpläne selbst werden jeweils durch den Federator des Systems an die zugehörige Datenquellen geschickt, dort ausgewertet und die erzeugten Ergebnisse anschließend an die zentrale Auswertungsstelle zurückgeliefert. Auch die Optimierung der föderierten Query findet bereits zentral statt. Die Output-Kardinalitäten der Operatoren eines an *src* auszuwertenden Teilplans p_{src} mit $src \in S_{fed}$ werden dabei anhand der Item-Counts der Datenstatistik $statistic(src) \in C_{dstats}$ abgeschätzt. Für die letztlich bestimmende Kosten-

schätzung von p_{src} wird eine Summe über den geschätzten Kardinalitäten aller enthaltenen Operatoren gebildet. Für die Kostenschätzung eines föderierten Plans insgesamt werden analog dann die Kostenschätzungen aller Teilpläne p_{src_i} zusammengeführt. Damit basiert die Kostenschätzung des gesamten Plans letztlich auf den Angaben, die durch die Datenstatistiken aller an der Auswertung beteiligten Datenquellen src_i gegeben sind.

Grundannahmen

Wie zuvor bereits angesprochen, besteht das Problem nun darin, dass die Datenbestände der Datenquellen in S_{fed} Veränderungen unterliegen, während die sie beschreibenden Datenstatistiken $statistic(S_{fed})$ von f unverändert weiterbestehen. Über einen gewissen Zeitraum, in dem eine Reihe von schreibenden Transaktionen auf dem Datenbestand einer Datenquelle src ausgeführt werden, die bestehende Daten verändern, entfernen oder neue Daten hinzufügen können, kann sich die tatsächliche momentane Datensituation (Item-Counts, Werteverteilungen, Abhängigkeiten) derart verändern, dass die Datenstatistik selbst „unzureichend“ ist. Das lässt sich auch wie folgt fassen:

Auf dem Datenbestand einer Datenquelle $src \in S_{fed}$ werden nun in einem bestimmten Zeitraum Δt mehrere schreibende Transaktionen δ_i sequentiell ausgeführt, die diesen jeweils in mehr oder weniger großem Umfang abändern:

$$D_{src}^{(0)} \xrightarrow{\delta_0} D_{src}^{(1)} \xrightarrow{\delta_1} D_{src}^{(2)} \xrightarrow{\delta_2} ... \xrightarrow{\delta_n} D_{src}^{(n+1)}.$$

Dabei bezeichnen $D_{src}^{(j)}$ jeweils den momentanen Datenbestand einer Datenquelle src zu Zustand j. Bezeichne außerdem $statistic(D_{src}^{(j)})$ die aktuelle Datenstatistik über dem momentanen Datenbestand $D_{src}^{(j)}$, wie sie sich bei einer Erhebung ergeben würde, und sei umgekehrt $D(statistic(src))$ mit $statistic(src) \in C_{dstats}$ der momentane Datenbestand, über dem die im Katalog vorgehaltene Datenstatistik $statistic(src)$ erhoben wurde.

Gilt nun allgemein $D(statistic(src)) \xrightarrow{\delta_1} ... \xrightarrow{\delta_n} D_{src}^{(j)}$, so kann man die Vermutung anstellen, dass mit wachsendem Zustandsabstand n auch $statistic(src)$ und $statistic(D_{src}^{(j)})$ selbst mit wachsender Wahrscheinlichkeit (nach einem bestimmten Maß) voneinander abweichen. Die im Katalog C_{dstats} für die Kostenschätzung vorgehaltenen Datenstatistik $statistic(src)$ müsste dann für ein bestimmtes n in der Folge als veraltet oder obsolet gelten. Diese Annahme liegt den weiter unten beschriebenen Verfahren zugrunde. Jedoch ist weder ein adäquates Maß für die Bestimmung einer solchen Abweichung sinnvoll zu

definieren, noch ist eine Abweichung zwischen den beiden Datenstatistiken in der Praxis direkt beobachtbar.

Die Lösungsidee, die im Folgenden verfolgt wird, stützt sich stattdessen auf die Beobachtung und Bewertung von Fehlern (nach einer definierten Metrik) der Kardinalitätsschätzungen von solchen Teil-Queries, die jeweils nur an einer Datenquelle ausgewertet und damit nur über die eine zugehörige Datenstatistik abgeschätzt werden. Dazu wird ebenfalls angenommen, dass mit zunehmender Verfälschung einer zugrunde liegenden Datenstatistik im Allgemeinen auch ein zunehmender Fehler (nach besagter Metrik) bei der Kardinalitätsschätzung zu erwarten ist.

4.2 Indikatoren für die Obsoleszenz verwendeter Datenstatistiken

Die Größe, die für die weitere Betrachtung grundlegend ist, ist die des relativen Fehlers einer Schätzung - hier der Kardinalitätsschätzung. Die Kardinalität, bzw. deren Schätzung, ist die Basis der Kostenschätzung, die bei der Suche eines optimalen Plans einer Query, z.B. durch Dynamic Programming, stattfindet. Abweichungen der dazu herangezogenen Datenstatistiken gegenüber der tatsächlichen Datensituation, die sich nach der oben formalisierten Vorstellung ergeben können, können problematisch sein. Im Rahmen einer automatischen Obsoleszenzbewertung der Datenstatistiken soll eine solche sich anbahnende Situation erkannt werden können. Analyseverfahren, die dies bewerkstelligen und gleichzeitig einfach durchzuführen sind, werden im Folgenden näher dargestellt.

Betrachtet man einen (Teil-)Plan $p_{src,i}$, der direkt an einer einzigen Datenquelle src ausgewertet wird, dann lässt sich der relative Fehler e der Kardinalitätsschätzung für $p_{src,i}$ mit der folgenden Metrik[1] quantifizieren:

$$error(p_{src,i}) = \frac{\left|(actual_{p_{src,i}} - estimated_{p_{src,i}})\right|}{estimated_{p_{src,i}}} \quad (estimated_{p_{src,i}} > 0). \tag{Gl. 4.2a}$$

oder mit $r_{src,i} = actual_{p_{src,i}}/estimated_{p_{src,i}}$ und $estimated_{p_{src,i}} > 0$:

$$e(p_{src,i}) = |r_{src,i} - 1|\,. \tag{Gl. 4.2b}$$

[1] Dieselbe Fehlermetrik kommt bereits in [IC91] zu Anwendung und wurde daraus übernommen.

Dabei bezeichnet $actual_{p_{src,i}}$ die tatsächliche und $estimate_{p_{src,i}}$ die zuvor geschätzte Output-Kardinalität des Wurzeloperators von $p_{src,i}$.

Bei der Beobachtung der zeitlich geordneten Ausführung von n Queries (bzw. deren Pläne) lässt sich nun eine Sequenz von quellenspezifische Fehlerbeobachtungen $\langle e_{src,i} \rangle$ erzeugen mit $e_{src,i} = error(p_{src,i})$ und $0 \leq i < n$. Diese Fehlersequenz lässt sich ebenso auch als Realisation eines Zufallsprozesses $(E_{src,i})$ mit $E_{src,i} = e_{src,i}$ betrachten, wobei über die stochastischen Eigenschaften der Zufallsvariablen $E_{src,i}$ (ohne weiteres) nichts Näheres bekannt ist. Dies wird im Weiteren zum Tragen kommen. Durch geeignete Verfolgung und Analyse von $\langle e_{src,i} \rangle$ sollte es möglich sein, auf eine mangelnde Aktualität (Obsoleszenz) einzelner Datenstatistiken zu schließen, um dann das Datenbanksystem dazu zu befähigen entsprechende Gegenmaßnahmen einzuleiten.

Die in den folgenden Abschnitten beschriebenen Verfahren sollen auf Basis aktueller Fehlersequenzen $\langle e_{src,i} \rangle$, die wie gesagt je einer Datenquelle $src \in S_{fed}$ zugeordnet sind, entscheiden, ob die zugehörigen Datenstatistiken $statistic(src) \in C_{dstats}$ tatsächlich als obsolet gelten müssen oder nicht. Die Verfahren realisieren je eine Entscheidungsfunktion φ, die je Datenquelle src mit $\langle e_{src,i} \rangle$ die Gültigkeit zweier sich gegenseitig ausschließender Annahmen $H0_{src}$ („Datenstatistik $statistic(src)$ aktuell“) und $H1_{src}$ („Datenstatistik $statistic(src)$ obsolet“) entscheidet. Sollte also im Laufe der Zeit für eine Datenstatistik $statistic(src)$ $H1_{src}$ als gültig erachtet werden, könnte anschließend eine Neuerhebung angestoßen werden.

if $\varphi(\langle e_{src,i} \rangle) = 0$ **then**
  $H0_{src}$ bestätigt
else if $\varphi(\langle e_{src,i} \rangle) = 1$ **then**
  $H1_{src}$ bestätigt
end if

Es folgt die Beschreibung der Entscheidungsverfahren, die dies nach verschiedenen Ideen realisieren. Die Implementation und die notwendige Evaluation werden in späteren Kapiteln diskutiert.

4.2.1 Test auf Einhaltung absoluter Fehlerschwellen

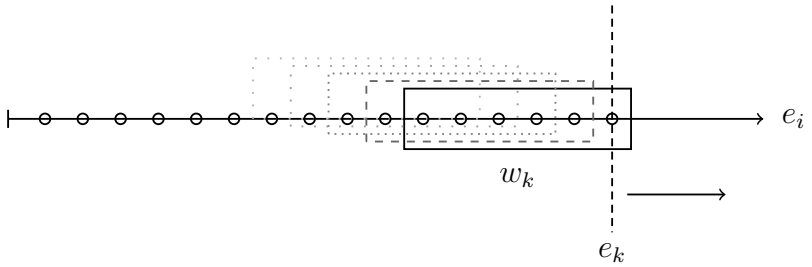
Die Grundidee für das hier beschriebene Verfahren ist, den zeitlichen "Trend"der Schätzfehler für je eine der Datenquellen daraufhin zu überprüfen, ob dieser eine gegebene spezifische Schwelle $t_{src} > 0$ $(src \in S_{fed})$ einhält, d.h. unterschreitet. Sollte zu einem bestimmten Zeitpunkt eine Überschreitung zu beob-

achten sein, wird unterstellt, dass die bisherige Datenstatistik $statistic(src) \in C_{dstats}$, auf der die letzten Schätzungen basierten, obsolet ist und durch eine dem aktuellen Datenbestand von src angemessene Datenstatistik ersetzt werden muss.

Ausgehend von der oben beschriebenen Fehlersequenz $\langle e_{src,i} \rangle$ wird eine einfache Trendabschätzung für die vorliegenden Schätzfehler vorgenommen. Zu diesem Zweck wird prinzipiell nach folgender Vorschrift ein gleitender Durchschnitt $\langle \bar{e}_{src,i,|w|} \rangle$ für die Fehlersequenz gebildet:

$$\bar{e}_{src,i,|w|} = \sum_{j=i-|w|+1}^{i} \frac{e_{src,i}}{|w|} \quad (i \leq |w| - 1).$$

Die *Fensterweite* $|w|$ gibt dabei vor, wie viele der Fehlerbeobachtungen $e_{src,i}$ aus der zugehörigen Sequenz für die Berechnung eines lokalen Durchschnittswerts $\bar{e}_{src,i,|w|}$ hinzuziehen sind. Praktisch lässt sich obige Rechnung damit auch mithilfe eines gleitenden Datenfensters w nachvollziehen, dessen aktuelle Ausprägung w_k nach Beobachtung eines Fehlers $e_{src,k}$ gerade die letzten $|w|$ Fehlerbeobachtungen $e_{src,k-|w|+1}$, $e_{src,k-|w|} \cdots$, $e_{src,k}$ enthält. Über diesen Werten wird dann der nach jeder Beobachtung aktuelle Durchschnitt $\bar{e}_{src,k,|w|}$ gebildet.



Ein zweiter Parameter, der *Schwellwertverktor* $\vec{t} = (t_{src_0}, t_{src_1}, ..., t_{src_n})^T$ ($\{src_0, src_1, ..., src_n\} = S_{fed}$), repräsentiert die Schwellwerte, die von den, den Datenquellen $src_0 ... \ src_n$ zugeordneten Fehlertrends einzuhalten sind. Aus praktischen Erwägungen, z.B. um allzu häufige Tests (und damit auch Indikationsmeldungen) zu vermeiden, findet die Aktualisierung des Datenfensters und die damit einhergehende Überprüfung erst nach mindestens $\lfloor |w| / 2 \rfloor$ neu vorliegenden Fehlerbeobachtungen statt. Anhand der Schwellwerte und der aktuellen Fehlertrends entscheidet das Verfahren dann für jede Datenquelle src dann die Gültigkeit von $H0_{src}$ und $H1_{src}$ nach der folgenden Vorschrift:

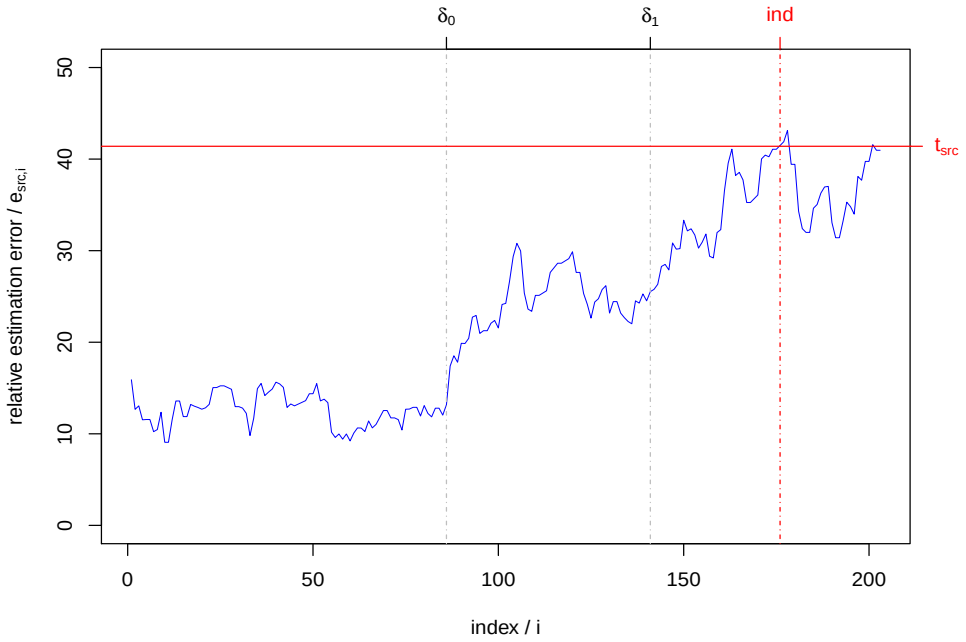


Abbildung 4.1: Illustration der Idee von Verfahren 1. Dargestellt ist der gleitende Durchschnitt der Fehlersequenz $\langle e_{src,i} \rangle$ mit einer bestimmten Fenstergröße. Nach Überschreitung der Fehlerschranke t_{src} (rot) an Stelle 176 findet eine Indikationsmeldung („ind“) statt. Der Fehler nimmt hier gemäß der Annahmen aus 4.1 nach Einbringung der Transaktionen δ_0 und δ_1 deutlich zu.

1. nehme $l = \lfloor |w| / 2 \rfloor$ neue Fehlerbeobachtungen $e_{src,k-l+1}$, $e_{src,k-l+2}$,..., $e_{src,k}$ in alte Fehlersequenz auf
2. berechne das neue Datenfenster w_k
3. entscheide über die Obsoleszenz einer Datenstatistik $statistic(src)$ mit

$$\varphi(\langle e_{src,i} \rangle) = \begin{cases} 0 & \bar{e}_{src,i,|w|} \leq t_{src} \\ 1 & \text{sonst} \end{cases}$$

Da das später implementierte Verfahren im laufenden Betrieb parallel zur Query-Auswertung arbeitet und daher ohne weiteres immer auf aktuellen Fehlerbeobachtungen arbeitet, ist die vollständige Berechnung von $(\bar{e}_{src,i,|w|})$ nicht notwendig. Es reicht aus, zu jedem Zeitpunkt je Datenquelle src auch nur den aktuellen Durchschnittswert $\bar{e}_{src,i,|w|}$ und die zugehörigen Fehlerwerte $e_{src,i-|w|+1}$, $e_{src,i-|w|} \cdots$, $e_{src,i}$ zu speichern. Bei Vorliegen einer neuen

Fehlerbeobachtung $e_{src,i+1}$ lässt sich dann sehr einfach der aktuelle Durchschnittswert $\bar{e}_{src,i+1,|w|}$ berechnen. Eine Illustration dieser Idee ist in Abbildung 4.1 gegeben.

Die Schwellwerte sind absolut und müssen für ein befriedigendes Verhalten des Verfahrens zunächst abgestimmt werden. In der weiter unten beschriebenen Evaluation der Gesamtlösung wird dies explizit vorgenommen (siehe Kapitel 6).

4.2.2 Vergleich benachbarter Fehlersequenzen

Eine andere Idee besteht darin, nicht auf die Einhaltung einer absoluten Fehlerobergrenze zu achten, sondern zu untersuchen, ob mögliche Veränderungen in der Häufigkeitsverteilung der Fehler auftreten. Betrachtet man mehrere separate, nicht überlappende Segmente einer gegebenen Fehlersequenz, so ließen sich durch geeignete statistische Signifikanztests solche Unterschiede zwischen einzelnen dieser Segmente oder Teilsequenzen (z.B. bezüglich der Verteilungslage der Fehlerwerte) feststellen.

Im Unterschied zum gerade beschriebenen Verfahren werden dazu entsprechend mehrere angrenzende Datenfenster verwaltet: Man betrachte eine Zerlegung einer quellenspezifischen Fehlersequenz $\langle e_{src,i} \rangle$ der Datenquelle src in $n+1$ zusammenhängende, nicht überlappende Teilsequenzen. Die Teilsequenzen seien, wie oben, ebenfalls in eigenen Datenfenstern $w_{i,j}$ $(0 \leq j \leq n)$ eingefasst. Der jeweilige Inhalt, das heißt die verwalteten Fehlerwerte, eines Fensters ist durch den Index i der letzten Fehlerbeobachtung und der Weite des Fensters bestimmt. Insgesamt sei dies wie folgt notiert:

$$\langle e_{src,i} \rangle = w_{i,0} \cup w_{i,1} \cup ... \cup w_{i,n}, \tag{Gl. 4.2.2a}$$

sodass

$$|\langle e_{src,i} \rangle| = |w_{i,0}| + n \cdot |w| \,. \tag{Gl. 4.2.2b}$$

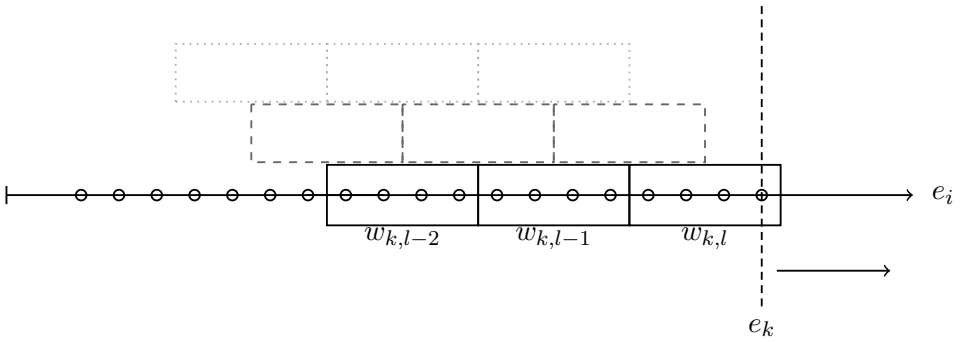
Dabei ist auch hier mit $|w|$ die Fensterweite gemeint, die hier für alle Fenster $w_{i,j}$ mit $j \geq 1$ gegeben ist. Fenster $w_{i,0}$ stellt hier nur sicher, dass die Zerlegung von $\langle e_{src,i} \rangle$ vollständig ist.

Für jede neu hinzukommende Fehlerbeobachtungen $e_{src,k}$ wird nun formal eine neue Zerlegung von $n+1$ Datenfenstern (nach Vorschrift Gl. 4.2.2a) gebildet. Praktisch lässt sich dies so realisieren, dass die Fenster $w_{k,j}$ mit $j \geq 1$ entlang der fortgesetzten Fehlersequenz nach „rechts“ verschoben werden und Datenfenster $w_{k,0}$ (gemäß Gl. 4.2.2b) entsprechend ausgeweitet wird. Sei

$w_{i,j}[x]$ die Fehlerbeobachtung an Stelle x $(0 \leq x < n)$ in $w_{i,j}$, dann gilt mit neu hinzukommender Fehlerbeobachtung $e_{src,k}$ daraufhin also:

- $w_{k,0} = w_{k-1,0} \cup \{w_{k-1,1}[0]\}$
- $w_{k,j} = w_{k,j} \cup \{w_{k-1,j+1}[0]\} \setminus w_{k-1,j}[0] \quad (0 < j < n)$
- $w_{k,n} = w_{k,n} \cup \{e_{src,k}\} \setminus w_{k-1,n}[0]$.

Eine solche Verschiebung wird, wie in 4.2.1, erst nach mindestens $\lfloor |w| / 2 \rfloor$ neuen Fehlerbeobachtung vorgenommen, sodass sich anschaulich das folgende Bild ergibt.



Unter der Annahme aus 4.1 gilt hier nun, dass sich die (auch zeitlich) separierten Teilsequenzen unter verschiedenen statistischen Aspekten unterscheiden, sofern in dem von diesen aufgespannten Zeitraum Datenveränderungen aufgetreten sind. Betrachtet man so zum Beispiel zwei Teilsequenzen der Länge $|w|$, von denen man weiß, dass die erste Schätzfehler über einer aktuellen und die zweite Schätzfehler über einer bekanntermaßen öbsoleten"Datenstatistik enthält, müssten sich also die Fehlerwerte beider Sequenzen statistisch deutlich voneinander absetzen. Diese Unterscheidung ist nach verschiedenen Gesichtspunkten möglich, die jeweils durch eigene Verfahren zu berücksichtigen sind. In diesem Abschnitt werden dazu zwei Verfahren vorgeschlagen. Das erste (Verfahren 2a) prüft mehrere Teilsequenzen paarweise darauf, ob deren Fehlermittelwerte übereinstimmen oder gegebenenfalls gegeneinander verschoben sind. Das zweite Verfahren(Verfahren 2b) prüft, ob die (geschätzte) Fehlerverteilung für Teilsequenz von der einer zweiten Teilsequenz abweicht oder nicht. Die Verfahren implementieren die Entscheidungsfunktion φ jeweils unter Anwendung eines statistischen Signifikanztests für den Vergleich zweier einander unabhängiger Stichproben. Da man wie bei Verfahren 1 jedoch auch hier an der Frage interessiert ist, ob gerade jetzt die aktuelle Datenstatistik als obsolet gelten muss oder nicht, ist eine der zu vergleichenden Stichproben immer die aktuellste Teilsequenz aus Datenfenster

$w_{i,n}$. Die andere Stichprobe wird aus einem zweiten Fenster $w_{i,u}$ entnommen, mit $0 < u < n$. Da es für die Wahl der zweiten Stichprobe also allgemein $n-1$ Möglichkeiten gibt, kommen auch $n-1$ verschiedene Testausführungen in Frage.

Bei statistischen Signifikanztests wird an gegebenen Daten geprüft, ob von zwei alternativen Modellannahmen oder Hypothesen (Nullhypothese, Alternativhypothese), die eine oder die andere gilt, wobei die Nullhypothese bevorzugt wird. Die Entscheidung über die Annahme einer der Hypothesen richtet sich nach einem vorzugebenden Signifikanzniveau α und einer über den Daten gebildeten Prüfgröße, die unter der Nullhypothese auf bekannte Weise verteilt ist. Das Signifikanzniveau gibt die noch akzeptierte Wahrscheinlichkeit dafür an, die wahre Nullhypothese durch Annahme der Alternativhypothese irrtümlicherweise abzulehnen. Wie bei Signifikanztests in der Praxis allgemein üblich, wird bei jeder Testausführung, anhand des konkreten Werts der Prüfgröße und der Prüfverteilung unter der Nullhypothese, der sogenannte p-Wert (p) berechnet [FKPT03, S. 417]. Dieser Wert gibt die Wahrscheinlichkeit wieder, den erhaltenen Prüfwert oder einen extremeren Wert unter der Annahme der Nullhypothese und der zugehörigen Prüfwertverteilung zu erhalten. Ist dann $p \leq \alpha$, wird die Nullhypothese zugunsten der Alternativhypothese abgelehnt, da der Prüfwert dann als nicht mit der Nullhypothese vereinbar gilt. Andernfalls wird die Nullhypothese angenommen.

Berechne nun $p_{\mathtt{X}}(w_{i,u}, w_{i,u})$ den p-Wert bei dem Vergleich zweier unabhängiger Stichproben aus den Datenfenstern $w_{i,u}$ und $w_{i,u}$ gemäß eines Signifikanztests `X` (siehe weiter unten). Die grundlegende Arbeitsweise beider Verfahren schlägt sich dann auf folgende Weise in der Realisierung der Entscheidungsfunktion φ nieder.

1. nehme $l = \lfloor |w| / 2 \rfloor$ neue Fehlerbeobachtungen $e_{src,k-l+1}$, $e_{src,k-l+2}$,..., $e_{src,k}$ in alte Fehlersequenz auf
2. berechne neue Datenfenster $w_{k,j}$ $(0 \leq j \leq n)$
3. entscheide über die Obsoleszenz einer Datenstatistik $statistic(src)$ mit

$$\varphi(\langle e_{src,i} \rangle) = \begin{cases} 1 & \exists w_{i,u}(u \in \{1, 2, ..., n-1\})(p_{\mathtt{X}}(w_{k,u}, w_{k,n}) \leq \alpha) \\ 0 & \text{sonst} \end{cases}$$

Nach jeder Verschiebung der Datenfenster um $\lfloor |w| / 2 \rfloor$, wird also in beiden Fällen geprüft, ob es mindestens eine zurückliegende Teilsequenz eines Fensters gibt, deren Werte sich nach Entscheidung des verwendeten Tests (abstrakt als `X` bezeichnet) unter dem jeweiligen Aspekt von denen der „jüngs-

ten“ Teilsequenz aus Fenster $w_{k,n}$ signifikant unterscheiden.

Um bei Verfahren 2a nun tatsächlich auf eine etwaige Lageverschiebung hin zu prüfen, bieten sich grundsätzlich mehrere Tests an, die sich auch in der Wahl des Lagemaßes unterscheiden. Ein parametrischer Test, der diese Fragestellung aufgreift, ist der Zwei-Stichproben-T-Test nach Welch[FKPT03, S. 455-456]. Geprüft wird dabei konkret, ob der (empirische) Mittelwert einer Stichprobe signifikant von dem einer zweiten abweicht. Der Nachteil dieses Tests ist, dass die Werte beider Stichproben als normalverteilt angenommen werden, was im vorliegenden Fall relativer Fehler nicht vertretbar ist. Stattdessen wird der nicht-parametrischer Mann-Whitney-Wilcoxon-Test [Pol97, S. 197 ff.] (durch p_{mww} in obigem Algorithmus) implementiert, bei dem nur angenommen wird, dass die Stichproben derselben Varianz unterliegen. Eine genaue Verteilung der Stichproben wird nicht explizit vorausgesetzt. Für die aus zwei Stichproben X, Y empirisch gewonnenen kumulativen Verteilungsfunktionen $Fn_X(x)$, $Fn_Y(x)$ liegt eine Lageverschiebung dann vor, wenn gilt: $\forall x : Fn_X(x) = Fn_Y(x + d)$ mit $d > 0$. Angewendet auf das hier vorliegende Problem wird durch Verfahren 2a überprüft, ob die beobachteten Fehler eines Datenfensters gegenüber denen eines zweiten Fensters allgemein (nach oben) verschoben sind.

In Verfahren 2b ist die Fragestellung, nach der zwei Stichproben bzw. Teilsequenzen aus zugehörigen Datenfenstern verglichen werden, etwas allgemeiner gefasst. Es werden auch hier die empirischen kumulativen Verteilungsfunktionen $Fn_X(x)$, $Fn_Y(x)$ zweier Stichproben X,Y vergleichend untersucht. Es wird geprüft ob diese identisch sind, oder ob sie sich unterscheiden, indem z.B. für ein x gilt $(F_X(x) > F_Y(x))$. Als konkretes Testverfahren wird hier der sogenannte Kolmogorow-Smirnow-Test (implementiert durch p_{ks}) eingesetzt, der genau das untersucht [MdS07, S. 201-202]. Es wird bei diesem Test jedoch nicht explizit untersucht, ob sich Unterschiede aufgrund einer Lageverschiebung zwischen den Verteilungen ergeben. Damit unterscheiden sich Verfahren 2a und 2b also auch in ihren Voraussetzungen, da Verfahren 2b auch auf Veränderungen der Varianz reagieren kann und damit allgemeiner anwendbar ist.

Bei beiden Verfahren möchte man feststellen, ob sich beobachtete Schätzfehler mit der Zeit zuspitzen, sich deren Verteilung gegenüber früheren Stichproben also „nach oben“ entwickeln. Aus diesem Grund realisieren sowohl Verfahren 2a wie Verfahren 2b ihre Untersuchungen in Form einseitiger (rechtsseitiger) Signifikanztests. Dies wird durch Halbierung der durch $p_{\mathtt{X}}$ jeweils gelieferten p-Werte und unter Beibehaltung des gewünschten Signifikanzniveaus α realisiert.

In Abbildung 4.2 werden beispielhaft die geschätzten Wahrscheinlichkeitsdichtefunktionen zweier Fehlerteilsequenzen gegenübergestellt. In der dargestellten Situation unterscheiden sich die Verteilungsfunktionen sowohl in der Lage ihrer Mittelwerte, als auch als auch in ihrer Form. Verfahren 2a und 2b sollten hier jeweils die zugrunde liegende Datenstatistik als „obsolet“ einschätzen können.

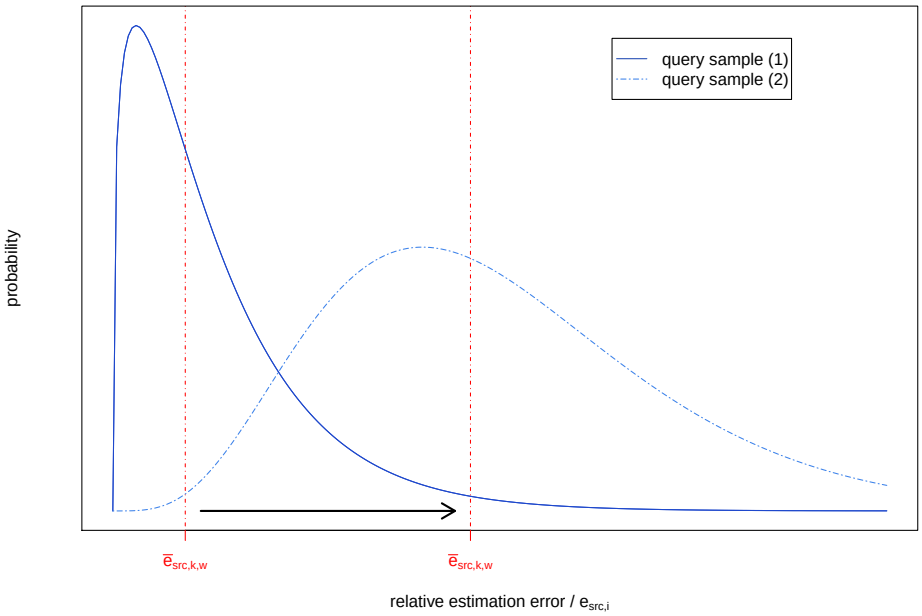


Abbildung 4.2: Idealisierte beispielhafte Darstellung der (geschätzten) Wahrscheinlichkeitsdichten zweier Fehlersequenzen bzw. -Stichproben aus zwei verschiedene Datenfenstern. Deutlich ist die Lageverschiebung der Mittelwerte sowie der Formunterschied zwischen den Verteilungen zu erkennen.

5 Instrumentation des Testsystems

Die im vorherigen Kapitel vorgenommene Modellierung für föderierte Datenbanksysteme im Allgemeinen (siehe Abschnitt 4.1) trifft insbesondere auf ein System zu, dass Thema dieses Abschnittes sein wird - der RDF-Federator. Um an diesem konkreten System unter der in Abschnitt 4.1 getroffenen Arbeitshypothese experimentelle Untersuchungen vornehmen zu können, sind einige Vorkehrungen zu treffen. Diese bestehen erstens in der Erweiterung des RDF-Federators, um die Beobachtung der Schätzvorgänge und die Dokumentation der dabei anfallenden Daten zu realisieren. Zweitens müssen anschließend die in Abschnitt 4.2 vorgeschlagenen Indikationsverfahren implementiert und in den erweiterten RDF-Federator eingebettet werden. All dies wird nun im Folgenden beschrieben.

5.1 Überblick

5.1.1 Der RDF-Federator

Das System, dessen Entwicklung in dieser Arbeit dargelegt wird, ist nicht isoliert zu betrachten, sondern ist Bestandteil eines RDF-basierten föderierten Datenbanksystems - der RDF-Federator[GS11]. Die Aufgabe dieses RDF-Federators besteht in der Integration von Linked-Data-Quellen. Die Datenquellen werden durch SPARQL-Endpoints gekapselt und als solche innerhalb des Systems katalogisiert. Die Aufgabe des RDF-Federators besteht nun darin, eine SPARQL-Abfrage, die zentral angenommen und geparst wird, in Teilabfragen aufzuteilen und diese an adäquate Quellen abzuschicken. Genaugenommen werden die Teilabfragen genau so erzeugt, dass sie auch (voraussichtlich) von mindestens einer Datenquelle beantwortet werden können. Die Entscheidung, ob eine Quelle adäquat ist, also der Beantwortung der betrachteten Teilabfrage dienen kann, wird anhand von Datenstatistiken über den einzelnen Quellen entschieden, die zusätzlich in einem zentralen Katalog verwaltet werden. Davon profitiert auch der kostenbasierte Optimizer, der vorab den optimalen Plan der gesamten föderierten Query sucht.

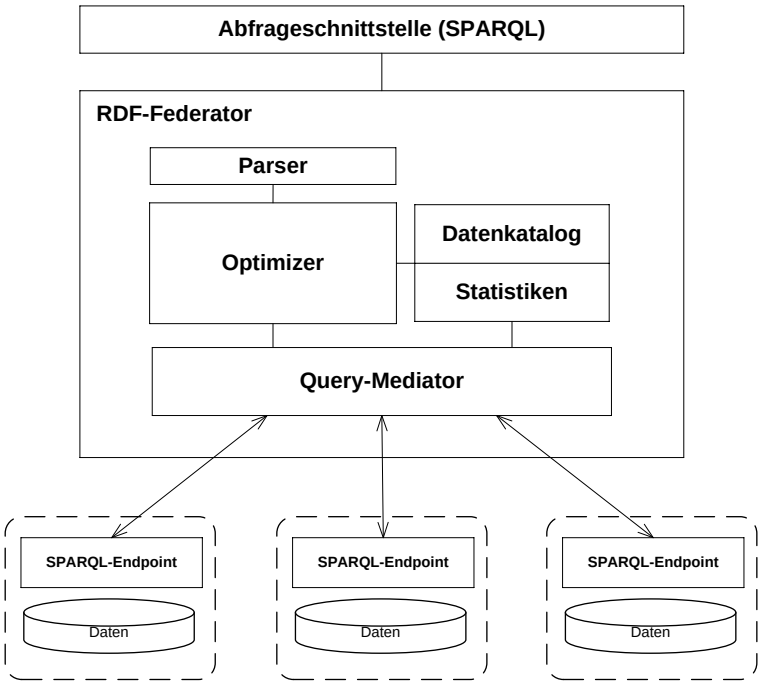


Abbildung 5.1: Grobarchitektur des RDF-Federators (angelehnt an [GS11]).

Architekturüberblick

Die Architektur des RDF-Federators[GS11] stellt sich wie folgt dar (siehe Abbildung 5.1).

Abfrageschnittstelle

Die Abfrageschnittstelle des RDF-Federators nimmt Abfragen entgegen. Diese sind in SPARQL formuliert und fragen im Allgemeinen Daten mehrerer Quellen an. In der der Arbeit vorliegenden Situation arbeitet der Federator innerhalb eines dazu eingerichteten Sesame-Triplestores, dem jedoch selbst keinerlei Anwendungsdaten überstellt sind. Die abzufragenden Daten werden stattdessen von mehreren Datenquellen bereitgestellt, die sich möglicherweise auch in entfernten Ausführungsumgebungen befinden. Mit den Mitteln von Sesame wird die Query übersetzt und zunächst logisch optimiert. Die soweit verarbeitete Query wird schließlich auch physisch optimiert. Dies wird vom Federation-Optimizer des Federators übernommen.

Federation-Optimizer

Der Federation-Optimizer, der zentrale Query-Optimizer des RDF-Federators, optimiert eine eintreffende föderierte Query nach Kostenschätzungen. Die Suche des günstigsten Plans richtet sich nach einer kostenbasierten Bewertung der alternativen Plankandidaten und wird mit Hilfe der Implementation eines Dynamic-Programming-Ansatzes realisiert. Für die Schätzung der Kosten eines Plans kommt ein Kostenschätzer zum Einsatz. Dieser zieht für seine Schätzungen selbst Kardinalitätsschätzungen der Ausgabeergebnisse einzelner Query-Operatoren hinzu. Diese Kardinalitätsschätzungen werden von einem zweiten Schätzer berechnet, der dazu direkt auf Datenstatistiken einzelner Datenquellen zurückgreifen kann. Aus demselben Grund muss auch bereits bekannt sein, an welchen Datenquellen eine Teil-Query schließlich ausgewertet werden muss. Auch diese Zuteilung von Query-Fragmenten findet anhand der Datenstatistiken statt.

Datenkatalog und -Statistiken

Der Federator verwaltet intern eine Sammlung von statistischen Zusammenfassungen bzw. Beschreibungen von Datenbeständen, die in den föderierten Datenquellen aufbewahrt werden. Dies sind Datenstatistiken, die in Form von Beschreibungen über dem Vocabulary of Interlinked Datasets[1](VoID) formuliert werden und sich jeweils auf eine der Datenquellen beziehen, deren Adressen ebenfalls vermerkt sind. In einer solchen Quellenbeschreibung oder Datenstatistik werden verschiedene „Item-Counts“ aufgeführt. Dazu zählen die Anzahl einzigartiger Tripel, Subjekte, Objekte und Properties im RDF-Graph der jeweiligen Quelle. Zudem werden je Property noch die Zahl der mit dieser in Verbindung stehenden Tripel, Subjekte und Objekte vermerkt. Auf dieser Basis lässt sich für Teile einer Query abschätzen, ob eine gegebene Datenquelle für deren Beantwortung in Frage kommt. Wie angesprochen, werden zudem über diesen Item-Counts im Laufe der Optimierung die Kardinalitäten der Teil-Ergebnisse und indirekt auch die Kosten eines Plans abgeschätzt.

Query-Executor/-Mediator

Durch den Query-Executor, oder auch Query-Mediator, wird ein Plan der Query an einer Reihe von Datenquellen schließlich ausgewertet. Dazu werden

[1]Eine Spezifikation ist hier zu finden `http://vocab.deri.ie/void/`

Teile der Query bereits in der Planungsphase durch die Adressen der für deren Beantwortung in Frage kommenden Datenquellen annotiert. Bei der Verarbeitung einer Query, verteilt der Query-Executor dann einzelne Teil-Queries an Datenquellen, wenn für diese mindestens eine Adresse annotiert ist. Das Abfrageergebnis einer solchen Teil-Query wird danach entsprechend an mögliche übergeordnete Teile der Query zurückgegeben, die dann erst zentral verarbeitet werden. Sollte eine Teil-Query an mehreren Datenquellen zu beantworten sein, werden die einzelnen Teilergebnisse zuvor vereinigt und erst anschließend weiterverarbeitet.

5.1.2 Die Systemerweiterung

Die softwaretechnische Umsetzung des in dieser Arbeit präsentierten Lösungskonzepts (siehe 4) besteht in der Ergänzung der Systemarchitektur des RDF-Federator durch weitere geeignete Teilkomponenten. Diese müssen im bestehenden RDF-Federator so in den Ablauf der Abfrageverarbeitung eingebunden werden, dass insgesamt eine geschlossene Feedback-Schleife (ähnlich der in Abbildung 3.1) zustande kommt. Für das konkret vorliegende Problem, angepasst an die Vorgaben des RDF-Federators, soll sich hierfür die in Abbildung 5.2 dargestellte Form ergeben.

Die Ergänzung wird durch ein Rahmenwerk eingebracht, das die nötigen Messungen bzw. Beobachtungen und eine anschließende Verarbeitung dieser erst ermöglicht. Dieses Rahmenwerk, im Folgenden als Observation-Framework bezeichnet, besteht aus einer Reihe von aktiven Systemkomponenten: dem FederationObserver, dem ObservationSequencer und einzelnen Indikatoren. Dem Framework liegt zudem ein eigenes Datenmodell zugrunde, in dem einzelne bei Verarbeitung einer konkreten Abfrage anfallende Beobachtungen zu einem Repräsentationsobjekt zusammengefügt werden können. Zu den Messgrößen, die an dieser Stelle betrachtet werden, gehören z.B. die Schätzungen für die Kosten und Kardinalitäten einer (Teil-)Abfrage, sowie deren tatsächliche Ergebniskardinalität und Auswertungsdauer (siehe 5.2). Der Vermerk dieser Größen wird durch das Datenmodell für jeden Operator der Query ermöglicht.

Der FederationObserver steht mit angepassten Versionen von Federation-Optimizer und Query-Mediator des RDF-Federators in Verbindung und nimmt von diesen jeweils anfallende Schätzungen und Messwerte entgegen. Die Werte werden, nachdem sie bereits in eine Repräsentation über dem Datenmodell eingefasst wurden, schließlich an den ObservationSequencer übergeben. Dieser verfolgt die Aufgabe aus einem der übergebenen Objekte jeweils mehrere

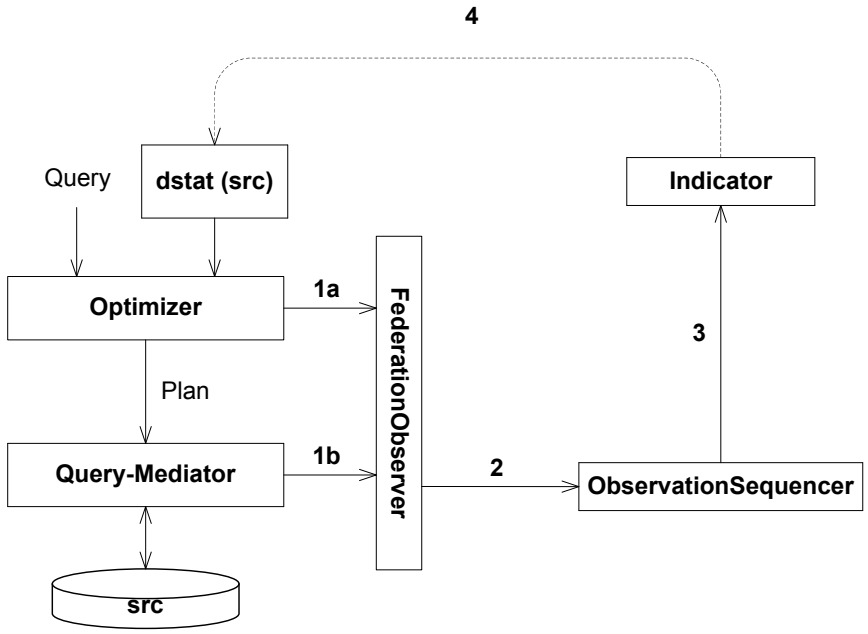


Abbildung 5.2: Illustration der umzusetztenden Feedback-Schleife. Die Daten, die während der Kostenschätzgun (1a) und Ausführung (1b) einer Query werden abgenommen. Der ObservationSequencer bereitet die Daten auf (2) und gibt sie an (mindestens) einen Indikator weiter (3). Unter Umständen wird die Obsoleszenz einer Datenstatistik (hier „dstat(src)“ genannt) erkannt und eine Neuerhebung dieser ausgelöst.).

Datenpunkte zu extrahieren. Diese setzten jeweils eine zusammenhängende Sequenz ähnlicher Datenpunkte fort. Den Indikatoren, die auf ebendiesen Sequenzen als Eingabe arbeiten, werden die erzeugten Datenpunkte durch den ObservationSequencer direkt übergeben. Gleichzeitig sorgt der ObservationSequencer für eine persistente Speicherung der besagten Datensequenzen, sodass sie nach Bedarf in ihrer Gesamtheit ausgelesen werden können.

Die einzelnen Indikatoren implementieren die in 4.2 vorgeschlagenen Verfahren und nehmen schließlich auf Basis der aktualisierten Datensequenzen für separate Datenstatistiken Obsoleszenzuntersuchungen vor und lösen gegebenenfalls für eine solche Datenstatistik eine vollständige Neuerhebung aus. In der Folge sollten Schätzfehler in Abhängigkeit von dieser neuen Datenstatistik wieder absinken.

5.2 Das Observations-Framework

Die Anpassung des RDF-Federators besteht aus softwaretechnischer Sicht, wie oben bereits erwähnt, in der Bereitstellung eines sogenannten Observations-Frameworks. Dieses sorgt für die systematische Erzeugung und Aufzeichnung von Beobachtungsdaten, die während einer konkreten Abfrageverarbeitung beobachtet werden. Es ist anzumerken, dass Observations-Framework insgesamt auf die möglichst vollständige Beobachtung von Queries ausgelegt ist, dies in der darauf aufbauenden Analyse (siehe unten) derzeit nur zum Teil ausgenutzt wird. Es sind aber weiterführende Entwicklungen vorstellbar, die die dargebotenen Daten vollständig ausnutzen. Im Folgenden wird eine weitergehende Darstellung der Framework-Bestandteile gegeben. Danach folgt die Beschreibung der Abläufe zwischen diesen, nach denen die Beobachtungen bei Auswertung einer Query gewonnen und verarbeitet werden.

Die Bestandteile

Observationsmodell

Das Observationsmodell ist ein erweiterbares Datenmodell, in dem die durch das Observations-Framework betrachteten Mess- bzw. Beobachtungsdaten abfrageorientiert eingefasst und zusammenfassend ausgedrückt werden können. Die Hauptmotivation dieses Datenmodells besteht in der von den konkreten Abläufen unabhängigen, vollständigen aber auch intuitionistischen, für sich selbst stehende Repräsentation von Beobachtungen an mehreren

Stellen einer Query. Gleichzeitig soll es sich leicht auf sich später ergebende Anforderungen anpassen lassen. Strukturell bildet es zu diesem Zweck den Operator-Baum eines Abfrageplans nach.

Konzeptionell unterscheidet das Modell dabei zunächst zweierlei:

- Observationsknoten
- Datenannotationen

Jeder der hier sogenannten Observationsknoten kapselt einen Operator einer Query. Dabei werden je nach Arität der berücksichtigten Operatoren auch bei den Observationsknoten entsprechende Subtypen unterschieden. Ein einzelner Observationsknoten kann dann selbst auf 0,1,2 oder N Kindknoten verweisen, die wiederum andere Query-Operatoren kapseln. Das insgesamt auf diese Weise zustande kommende Gebilde ist ein Baum, der (weitestgehend) mit dem Operator-Baum der beobachteten Query deckungsgleich ist (ein Beispiel findet sich in Abbildung 5.3).

Jeder Observationsknoten kann nun durch typisierte Datenannotationen annotiert werden. Durch Belegung entsprechender Referenzen, die von einem einzelnen Knoten verwaltet werden, werden diese dem jeweiligen Knoten angeheftet. Zu den derzeit explizit im Observationsmodell durch spezielle Klassen abgedeckten Annotationen gehören:

- die Schätzung der Kardinalität
- die reale Ergebniskardinalität
- der resultierende relative Fehler nach Gl. 4.2a
- die Schätzung Ausführungskosten
- der Startzeitpunkt der Auswertung
- die vollständige Antwortzeit (als Kostenäquivalent)

Konkrete Instanzen dieser Annotationen beziehen sich nur auf Teilabfragen der insgesamt beobachteten Query. Dazu werden sie genau dem Observationsknoten zugeordnet, der den Wurzeloperator der Teilabfrage abbildet.

FederationObserver

Die zentrale Komponente, an der sämtliche Beobachtungsvorgänge zusammenlaufen, ist der FederationObserver. Die Aufgabe des FederationObservers besteht in der Verarbeitung und Weitergabe von Ausdrücken über dem Observationsmodell. Der FederationObserver, der dem RDF-Federator direkt unterstellt ist, interagiert bei der Auswertung einer Query mit Kosten-

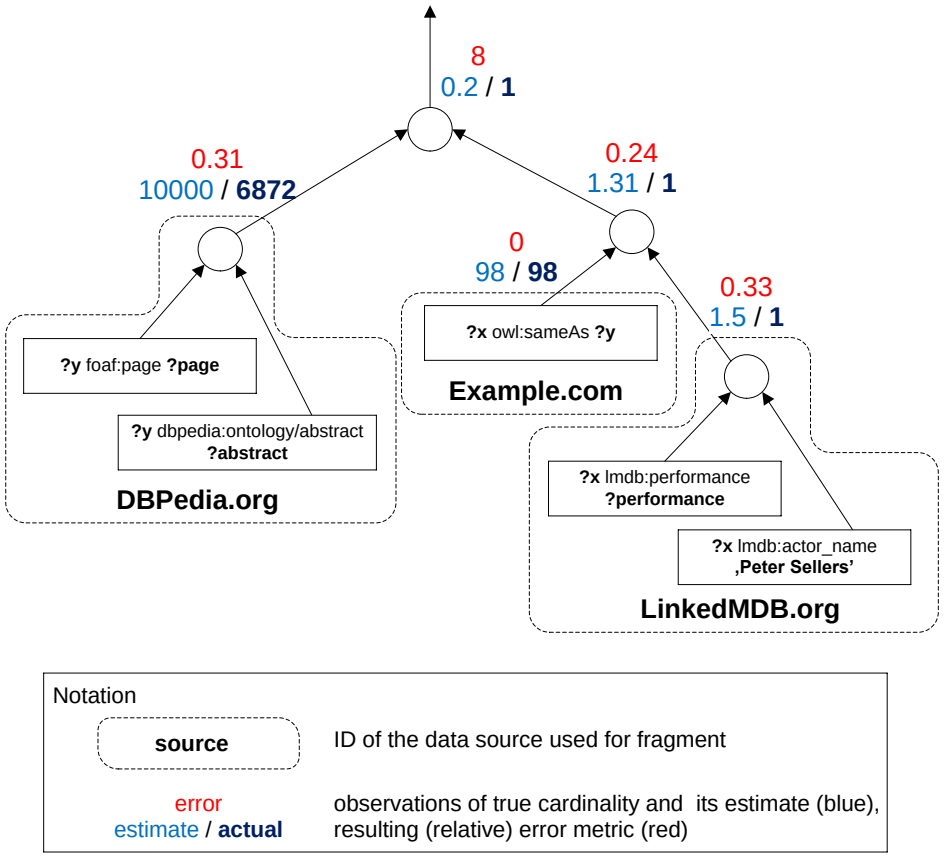


Abbildung 5.3: Beispiel eines Observationsobjekts. Gekapselt wird der algebraische Ausdruck aus 2.3. Der Übersicht halber werden nur hier einige (beispielhafte) Annotationen dargestellt.

und Kardinalitätsschätzer und dem Query-Executor des Federators. Sowohl die eingesetzten Schätzer, als auch der Query-Executor wurden so angepasst, dass entsprechende Beobachtungen erst explizit abzugreifen sind. Für die Beobachtung der tatsächlichen Auswertungszeiten und Ergebniskardinalitäten wurde ein angepasstes Iterator-Konzept entworfen, das innerhalb des Query-Executors zur Verwendung kommt. Im Laufe des Auswertungsvorgangs werden Messungen der Ergebnisumfänge und Auswertungszeiten durch die auszuschöpfenden Iteratoren selbst vorgenommen und für die Herstellung der Beobachtungsobjekte weitergegeben.

ObservationProcessor, ObservationSequencer und SequenceStore

Die Aufgabe eines ObservationProcessors ist es, ein einzelnes Observationsobjekt, dass von dem FederationObserver aus der Verarbeitung einer Query gewonnen wird, anzunehmen und nach bestimmten Maßgaben weiterzuverarbeiten. Bei der Instanziierung des FederationObserver wird dazu auch eine Instanz eines solchen ObservationProcessors erzeugt und registriert. Bei dem ObservationProcessor handelt es sich zunächst nur um eine abstrakte Schnittstelle, die von zwei alternativen Komponenten implementiert werden kann. Zum einen sorgt der ObservationSerializer für die Möglichkeit ein übergebenes Observationsobjekt in einen fortgeführten Objektstrom zu serialisieren. Dadurch ist die Möglichkeit gegeben, mehrere Beobachtungen in Dateien aufzuzeichnen und erst später weiterzuverarbeiten. Die zweite implementierende Komponente ist der ObservationSequencer. Dieser nimmt ebenso Observationsobjekte vom FederationObserver entgegen und erzeugt aus den annotierten Messdaten Datenpunkte fortlaufender Datensequenzen, die sich im laufenden Betrieb schließlich analysieren lassen. Der ObservationSequencer betrachtet dabei nur die Teil-Queries, die direkt an einer Datenquelle ausgewertet werden und damit auch über deren Datenstatistik abgeschätzt werden. In einer föderierten Query können mehrere solche Teil-Queries enthalten sein. Um die erzeugten Zeitreihen persistent und komfortabel aufzubewahren, übergibt der ObservationSequencer seine erzeugten Datenpunkte einem SequenceStore. Dies ist ein gekapseltes Repositorium, in dem mit einem sehr einfachen Datenschema Folgen (Sequenzen) zusammengehörigen Datenpunkten abgespeichert werden können. Ein weiterer Vorteil, der sich durch den Einsatz eines solchen SequenceStores ergibt, liegt darin, dass sich die darin verwalteten Beobachtungsdaten unter Verwendung des SPARQL-Interfaces von entfernter Stelle z.B. zum Zweck des Debuggings auslesen und gegebe-

nenfalls visualisieren lassen.

Indikatoren und Meldemechnismus

Ziel der gesamten hier dargestellten Lösung ist letztlich der „rechtzeitige“ Anstoß von ansonsten sehr teuren Neuerhebungen einzelner Datenstatistiken. Die Daten, die im Laufe der Zeit durch den ObservationSequencer erzeugt werden, werden dazu von einem (oder mehreren verschiedenen) im laufenden Federator betriebenen Indikatoren auf Hinweise untersucht, die gegebenenfalls darauf hindeuten, dass eine Datenstatistik einer föderierten Datenquelle nicht mehr mit deren Datenbestand in Einklang zu bringen ist, also obsolet ist. Die vorgeschlagenen Verfahren wurden in Abschnitt 4.2 bereits formal dargestellt. Auf softwaretechnischer Ebene werden sie durch Implementationen einer sogenannten Indicator-Schnittstelle realisiert. Eine Instanz eines solchen Indikators arbeitet auf Datensequenzen, wie sie der ObservationSequencer ausgibt. Sobald ein Indikator-Verfahren darauf hindeutet, dass eine Datenstatistik obsolet ist, wird durch den Indikator eine Indikationsmeldung ausgegeben, die dann eine Neuerhebung der Datenstatistik nach sich ziehen kann.

5.2.1 Beobachtung bei der Query-Verarbeitung

Eine SPARQL-Query, die von dem Federator entgegengenommen wird, passiert mehrere Verarbeitungsschritte, an denen nun auch Komponenten des Observations-Frameworks beteiligt werden - eine abstrakte Darstellung dieses Vorgangs ist in Abbildung 5.4 gegeben.

Als Erstes wird die SPARQL-Query, wie bereits erwähnt, durch Sesame-eigene Mittel übersetzt und logisch optimiert. Anschließend stellt der Optimizer des Federators einen Plan her, der basierend auf den zentralen Datenstatistiken der beteiligten Datenquellen als optimal angesehen wird. Während der Optimierung und nachfolgenden Auswertung der Query wird im FederationObserver eine neue Beobachtungssitzung gestartet, die erst mit der Erschöpfung der später erzeugten Ergebnis-Iteratoren wieder beendet wird. Bei der Optimierung der Query arbeitet der Query-Optimizer des RDF-Federators selbst nur auf einer Abstraktion des Sesame-eigenen Query-Models. Dieses deckt im Wesentlichen nur zwei Operatoren, Triple-Patterns und deren Joins, ab. Während im Laufe der Query-Planung mögliche Pläne aufgezählt werden, kommt auch bereits die Kostenschätzung zum Einsatz,

auf deren Grundlage suboptimale Pläne verworfen werden. Der Kostenschätzer des Federators arbeitet dazu ebenso auf demselben vereinfachten Query-Modell. Der Schätzer traversiert dabei den entsprechenden Operator-Baum zunächst top-down von der Wurzel des Plans zu den Blättern, das heißt hier den Triple-Patterns. Anschließend werden Kosten bei der Rückwärtsverfolgung bottom-up von den Blättern her aufsummiert. Dabei entsprechen die Kosten eines Joins, bzw. dessen Anwendung, hier vereinfacht der Summe seiner Input-Kardinalitäten, d.h. der Output-Kardinalitäten seiner Kind-Knoten. Um diese Kardinalitäten abzuschätzen, greift der Kostenschätzer auf den Kardinalitätsschätzer zurück, der die Output-Kardinalität eines Operators nach einem ähnlichen Traversierungsmuster von den Blättern zur Wurzel hin abschätzt.

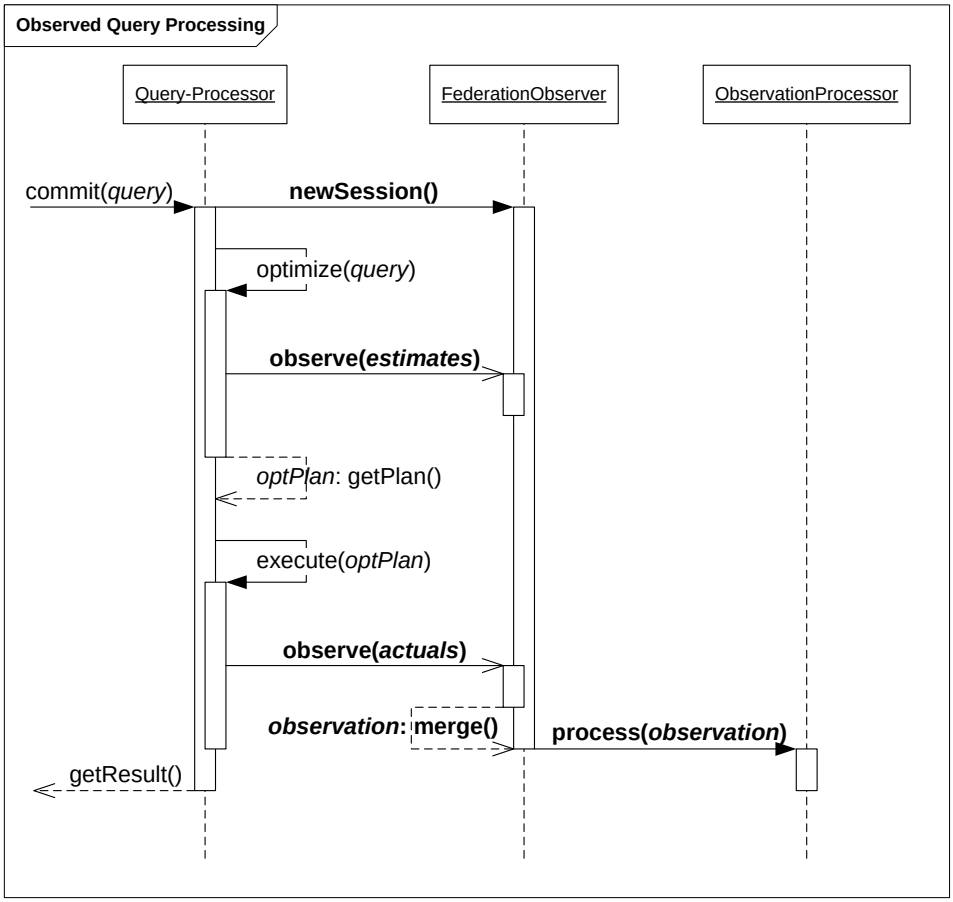


Abbildung 5.4: Ablauf der in das Query-Processing eingearbeiteten Observation.

Für die Beobachtung dieser beiden Werte für je einen Operator des Operator-Baums, kommen angepasste Versionen der beiden Schätzer zum Einsatz.

Während der top-down Traversierung wird dazu jeweils sukzessive ein Observationsbaum erzeugt, der strukturell mit der Query selbst übereinstimmt. Bei der Schätzung der Kosten bzw. Kardinalitäten bottom-up, werden die Schätzwerte an den zugehörigen Observationsknoten abgetragen.

Für die Auswertung des schließlich optimierten Plans, für den also bereits die Kosten- und Kardinalitätsschätzungen bekannt sind, wird in einem ganz ähnlichen Verarbeitungsvorgang ein Iterator-Baum erzeugt, der in seiner Struktur dem Operator-Baum nachempfunden ist. Ein einzelner Iterator ist ein Abstraktionsobjekt für die Anwendung eines physischen Operators der Query, das unter anderem auch bei Sesame Verwendung findet. Ein Iterator verarbeitet für die Ausgabe der eigenen Ergebnisse die Ausgabe von Kind-Iteratoren. Auch für die Erzeugung eines auf diese Weise geschachtelten Ergebnis-Iterators findet eine top-down Traversierung des Operator-Baums der auszuführenden Query statt. Bei der Rückwärtsverfolgung, findet die eigentliche Konstruktion des Iterator-Baums statt.

Diese Auswertungsstrategie wird für das Observations-Framework nun wie folgt angepasst. Bei der zwischenzeitlichen Erzeugung eines Iterators, der das Ergebnis eines besuchten Operators berechnen soll, wird dieser durch einen speziellen Iterator, einen so bezeichneten NotifyingResultWrapper, gekapselt. Ein solcher wird, wie der ursprüngliche Iterator, übergeordneten Iteratoren als Kind-Iterator dienen. Bei Abfrage der Ergebnistupel, bedient der Wrapper dann lediglich den beinhalteten Iterator, zählt dabei jedoch den Ergebnisumfang mit und vermerkt diese intern. Gleichzeitig werden der Zeitpunkt für die erstmalige Anforderung eines Tupels, dessen Antwortzeit, und die Dauer für die vollständige Ausschöpfung des Teilergebnisses intern gespeichert. Liefert der beinhaltete Iterator kein weiteres Ergebnistupel mehr, so werden die erhobenen Messwerte dem strukturell zugehörigem Observationsknoten eines neuen Observationsbaums eingetragen, der parallel zur Erzeugung des Iterator-Baums ebenfalls angelegt wurde. In dem Fall, in dem auch der die Wurzel-Iterator keine weiteren Tupel liefert, wird schließlich der dann vollständig besetzte Beobachtungsbaum an den FederationObserver weitergegeben.

Auf Seite des FederationObservers sind in der aktuellen Beobachtungssitzung dann zwischenzeitlich also drei Observationsbäume eingetroffen. Da diese jedoch der gerade zu verarbeitenden Query nachempfunden sind, unterscheiden sie sich strukturell nicht. Der FederationObserver überlagert die Bäume, bzw. führt die annotierten Daten auf einem Baum zusammen. Der resultierende Observationsbaum wird schließlich an eine QueryProcessor-Instanz übergeben. Im einfachen Fall kann diese Instanz z.B. ein Observa-

tionSerializer sein, der die übergebenen Bäume nacheinander in einer Datei serialisiert. Wichtiger ist der ObservationSequencer, dessen Funktionen im Folgenden dargestellt wird.

Wichtiger ist der ObservationSequencer, dessen Funktionen im Folgenden dargestellt wird.

5.2.2 Sequenzierung der Beobachtungen

Bei der Sequenzierung von Beobachtungen werden dem ObservationSequencer von dem FederationObserver erzeugte Ausdrücke des Observationsmodells übergeben. Die Aufgabe des ObservationSequencer ist nun, wie bereits angedeutet, die Herstellung bzw. Fortführung mehrerer Datensequenzen. Diese sind jeweils genau einer der föderierten Datenquellen zugehörig. Eine Query, die an den RDF-Federator gestellt wird, ist an mindestens eine, im Allgemeinen jedoch mehrere, Datenquellen gerichtet ist. Daher entstehen bei der Sequenzierung eines Beobachtungsobjekts einer einzigen Query auch entsprechend mehrere Datenpunkte. Die bereits bestehenden Datensequenzen der beteiligten Datenquellen können dann durch diese Datenpunkte weiter fortgesetzt werden.

Ein konkreter Observationsbaum, der dem ObservationSequencer durch den FederationObserver übergeben wird, wird zunächst in Teilausdrücke aufgetrennt, die sich bereits nur noch auf eine Datenquelle beziehen. Dazu wird der Baum von der Wurzel aus soweit traversiert, bis ein dabei gerade betrachteter Observationsknoten exakt eine Datenquellenannotation aufweist. Dies bedeutet dann, dass die Teilabfrage ab dieser Stelle nur an dieser einen Datenquelle ausgewertet wurde. Der Knoten wird vermerkt und die Traversierung an anderer Stelle fortgesetzt. Nach Abschluss der Traversierung liegt demnach eine Menge von Observationsknoten vor. Jeder einzelne dieser Observationsknoten trägt also Informationen, die an der Wurzel einer vollständig an entfernter Stelle ausgewerteten Query abgegriffen wurden. Für jeden betrachteten Observationsknoten werden anschließend sämtliche Annotationen ausgelesen und zusammen mit der vermerkten Adresse zu einem gemeinsamen Datenpunkt zusammengeführt. Die einzelnen Annotationen stellen damit also Komponenten eines solchen Datenpunktes dar.

Formal hat ein solcher Datenpunkt p die Form:

$$p = (i, src, est, act, e, c, t, dt, q, jc)$$

Darin enthalten sind:

1. Sequenznummer i der Query
2. Datenquellen-Adresse src
3. geschätzte Kardinalität est
4. tatsächliche Kardinalität act
5. errechneter relativer Fehler e
6. geschätzte Ausführungskosten c
7. der Startzeitpunkt t der Auswertung
8. die vollständige Antwortzeit dt (als Kostenäquivalent)
9. Stringrepräsentation q der Query
10. Anzahl der beinhalteten Joins jc.

Die Komponenten 2. - 8. entspricht also den oben erwähnten Annotationen, die für jeden der Observationsknoten vorliegen. Nach dieser Formalisierung bilden die Datenpunkte genau genommen eine einzige fortgesetzte Datensequenz. Diese wird im Rahmen der Analyse jedoch über die Datenquellen-Adressen src wieder in die besagten, einzelnen Datenquellen zugehörigen, (Teil-)Sequenzen zerlegt wird. Bei dieser Analyse (bzw. deren gegenwärtigen Implementation) werden nur die Komponenten src, i und e betrachtet. Jedoch können auch die anderen Komponenten z.B. für mögliche Weiterentwicklungen des Systems von Bedeutung sein (für das Debugging während der Entwicklung des Systems, aber auch bei der unten beschriebenen Evaluation waren diese bereits hilfreich).

Die nach diesem Schema aufgebauten Datenpunkte, die nach jeder Beobachtung einer Query durch den ObservationSequencer aufgebaut werden, werden wie bereits erwähnt an die dem Sequencer angeschlossenen Indikatoren übergeben. Weiteres dazu im folgenden Abschnitt.

Zur Speicherung der Datensequenzen kommt ein sogenannter SequenceStore zum Einsatz. Dieser kapselt ein lokales Sesame-Repository, in dem die Datenpunkte nach einem sehr einfach strukturierten Datenschema aufbewahrt werden können. Nach Bedarf können mehrere oder alle Datenpunkte zu beliebiger Zeit auch wieder ausgelesen werden. In der vorliegenden Arbeit wird dies für die weiter unten diskutierte Evaluation ausgenutzt. Es sind aber auch Weiterentwicklungen des Observation-Frameworks denkbar, um von entfernter Stelle z.B. zu Debug-Zwecken auf eine solche SequenceStore-Instanz zuzugreifen.

5.2.3 Verarbeitung der Datensequenzen

Wie schon erwähnt, werden die drei Indikationsverfahren aus dem vorangegangen Kapitel durch spezielle Implementationen einer Indicator-Schnittstelle realisiert. Diese sind im Einzelnen:

- ErrorTresholdIndicator (Indikationsverfahren 1 (4.2.1))
- MWWIndicator (Indikationsverfahren 2a (4.2.2))
- KSIndicator (Indikationsverfahren 2b)

Als Indicator implementiert jeder dieser Indikatoren jeweils eine process-Methode. Über Aufrufe dieser Methode wird jedem Indikator, der beim ObservationSequencer registriert ist, jeweils ein einzelner neu erzeugter Datenpunkt übergeben. Die Indikatoren sind jeweils für die Verwaltung ihrer Datenfenster (siehe Kapitel 4.2) selbst verantwortlich und nehmen die übergebenen Datenpunkte nach einer einfachen Prüfung in diese auf. Ein wichtiger Teil dieser Überprüfung ist eine (sehr) einfache Behandlung von Ausreißern bei den Fehlerwerten, da diese bei der betriebenen Analyse problematisch werden könnten. Dabei werden einzelne Datenpunkte nur dann akzeptiert, wenn der beinhaltete Fehlerwert in einem vorgegebenen Bereich liegt (gute Ergebnisse bei der Evaluation lieferte z.B. eine Bereichsfilterung mit $1 < e_i < 2500$). Für jede (bisher bekannte) Datenquelle verwaltet jeder Indikator separate Datenfenster. Die dazu nötige Zuteilung der Datenpunkte nach deren Zugehörigkeit zu einer Datenquelle, findet wie beschrieben über die vermerkte Quellenadresse *src* statt.

Wie im Abschnitt 4.2 beschrieben, werden die in den aktuellen Datenfenstern enthaltenen Daten nach den bereits beschriebenen Verfahren analysiert. Wenn als Ergebnis einer solchen Analyse eine Indikationsmeldung auszugeben ist, ruft der Indikator eine indicateObsolescence-Methode auf, die von Instanzen der sogenannten IndicationListener-Schnittstelle implementiert werden (vgl. Abbildung 5.3). Als Argumente werden dabei die Adresse der betroffenen Datenstatistik und die Sequenznummer der zuletzt betrachteten Query übergeben. Im hier betrachteten Fall dient der FederationObserver selbst als ein solcher IndicationListener und steht dazu mit den betriebenen Indikatoren in einer Subscriber-Listener-Beziehung. Etwaige Neuerhebungen wären an dieser Stelle in die Wege zu leiten. Die genaue Form dieses Schritts war für die vorliegende Arbeit jedoch nicht direkt von Interesse. Es wäre auch denkbar Indikationsmeldungen einem menschlichen Betreiber nur anzuzeigen. Die eigentliche Neuerhebung betroffener Datenstatistiken würde dann ggf. manuell durchführt.

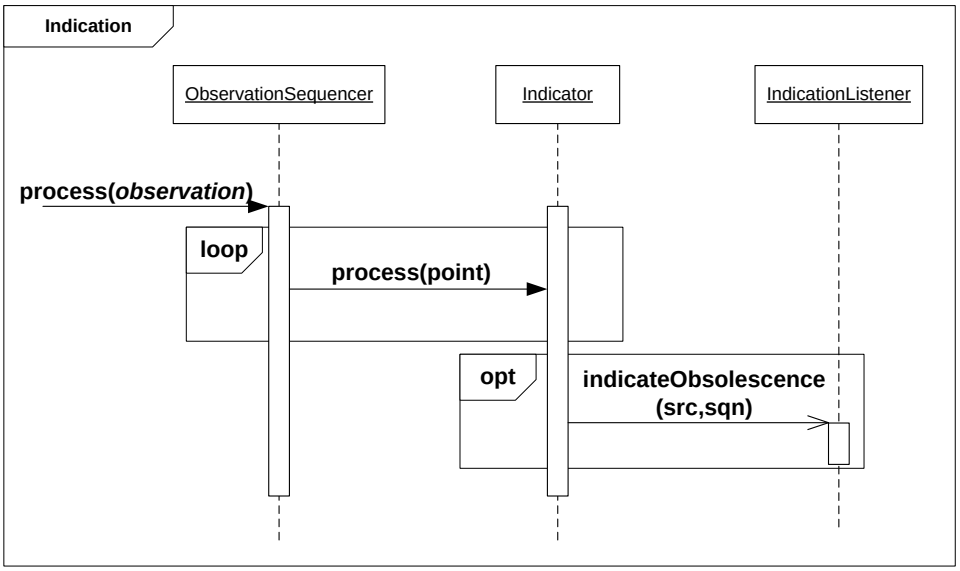


Abbildung 5.5: Ablauf der Observationsverarbeitung durch die betriebenen Indikatoren.

Für die im nun folgenden Kapitel diskutierte Evaluation wird hier jedoch ein automatischer Austausch der betroffenen Datenstatistik durch eine weitere bereits vorbereitete, an die Daten optimal angepasste Datenstatistik vorgenommen.

6 Evaluation

In einem praktischen Einsatzszenario für das bereits dargestellte föderierte Datenbanksystem (siehe 5.1.1) muss man davon ausgehen, dass sich die Datenbestände, über die zwecks Abfrageoptimierung zentrale Datenstatistiken verwaltet werden, verändern. Dadurch bilden ebendiese Statistiken die momentane Datensituation möglicherweise so schlecht ab, dass eine Query nicht optimal verplant wird. Die in den vorherigen Kapiteln vorgestellte Lösung soll die Erkennung solcher Situationen ermöglichen und das Datenbanksystem (siehe Kapitel 5.1.1) zur Erhebung neuer, angemessener Datenstatistiken bewegen können. Um dies empirisch zu überprüfen, wurde eine Evaluation der Lösung nach verschiedenen Gesichtspunkten vorgenommen. Im Folgenden werden die Planung und Durchführung dieser Evaluation beschrieben und die dabei gewonnen Ergebnisse diskutiert und interpretiert.

6.1 Untersuchungsziele

Die hier diskutierte Evaluation dient der Klärung mehrerer Fragestellungen, die sich für die zuvor diskutierten Verfahren und deren Implementation und Einbettung in dem betrachteten RDF-Föderationssystem ergeben. Diese sind im Einzelnen:

1. Sind die Annahmen, die den Verfahren aus Abschnitt 4.2 zugrunde liegen plausibel? Sind also Schätzfehler dann in größerer Ordnung zu beobachten, wenn die für die Kardinalitätsschätzung herangezogenen Datenstatistiken verfälscht sind? Wie verändert sich die Verteilung auftretender Fehler in Abhängigkeit von der Verfälschung der Datenstatistik?
2. Sind die Verfahren grundsätzlich geeignet, zunehmend verfälschte Datenstatistiken zu erkennen? Das heißt, ist die Testgrundlage jedes der einzelnen Verfahren, nach der diese Erkennung jeweils realisiert werden soll, überhaupt gegeben?
3. Wie sensibel reagieren die im Testsystem implementierten Indikationsverfahren auf beobachtete Schätzfehler und wie entwickeln sich die Schätzfehler mit Betrieb der Indikatoren und etwaigen Neuerhebungen der Datenstatistiken?

4. Können durch Einsatz der einzelnen Verfahren Kosten eingespart werden, die ohne Aktualisierung verfälschter Datenstatistiken aus einhergehenden Fehlschätzungen und suboptimal erzeugten Plänen möglicherweise erwachsen?

Diese Fragestellungen werden nun in den folgenden Abschnitten detailliert aufgegriffen. Eine zusammenfassende Diskussion der Ergebnisse findet sich schließlich in Kapitel 6.5.

6.2 Testumgebung und Datensituation

Die im Weiteren beschriebenen Tests wurden an einer Instanz des in Kapitel 5.2 beschriebenen angepassten RDF-Federators ausgeführt. Der Rechner, an dem diese Tests stattfanden, besitzt die folgenden Leistungsmerkmale:

- CPU: AMD Athlon II x4, 4 Kerne mit je 3.0 GHz
- RAM: 4 Gigabyte
- Betriebssystem: Windows 7 Professional (64-bit)

Die entwickelten Verfahren nehmen für jede der an der Föderation teilnehmenden Datenquelle eine Fehlerbeobachtung und -Analyse separat vor, um die Verfälschung der einer Quelle jeweils zugeordneten Datenstatistik einschätzen zu können. Die verteilten Queries, die an den RDF-Federator gestellt werden, werden zu diesem Zweck explizit in ihre Quellen-bezogenen Teilqueries zerlegt. Um nun im Weiteren eine unnötige Komplexität der Evaluation zu vermeiden, wird die Föderation nur einer Datenquelle betrachtet. Die grundsätzliche Funktionalität der Verfahren sollte sich dann auch so zeigen lassen. Der RDF-Federator wird daher auf nur eine lokale Datenquelle angewendet, die in Form eines RDF-Repositories einer lokalen Sesame-Serverinstanz vorliegt. Die im Laufe des Testbetriebs anfallenden Beobachtungsdaten werden im SequenceStore persistent eingelagert und für die folgende Evaluation ausgelesen und geeignet aufbereitet. Etwaige Indikationsmeldungen, die im Betrieb auf Basis derselben Beobachtungen von den Indicator-Instanzen des Federators gemeldet werden können, werden separat in Log-Dateien verzeichnet und später ebenfalls betrachtet

Die Datenquelle, die vom Testsystem angebunden wird, beinhaltet eine vorbereitete Menge an RDF-Tripeln. Diese RDF-Daten, die für das Szenario als Grundlage dienen sollen, stammen selbst aus einem Ausschnitt des DBPedia-

Datensatzes, wie er bereits in [HMZ10] für Benchmarks Verwendung findet[1]. Der Ausschnitt ist ein Teilgraph, der von circa 43,6 Millionen RDF-Triples aufgespannt wird und einen Datenumfang von etwa 6 GByte besitzt.

Es gibt mehrere Gründe für die Wahl dieses Datensatzes für die folgende Evaluierung. Gerade DBPedia ist eine besonders große und populäre Datenquelle der LOD-Cloud, eine Evaluierung mit realistischen Daten und Queries dieser Quelle ist daher entsprechend gegeben. Zudem werden für die Evaluierung viele SPARQL-Queries benötigt, da die beschriebenen Verfahren erst durch Beobachtungsreihen größeren Umfangs statistisch relevante Schlussfolgerungen ableiten können. Das heißt es müssen viele Queries aus einer heterogenen Menge entnommen und an dem Testsystem sequenziell ausgewertet und beobachtet werden. Für DBPedia liegen auch unter diesem Gesichtspunkt genügend Daten vor, aus denen geschöpft werden kann: Im Rahmen der USEWOD2011 wurden Server-Logs (u.a.) der DBPedia ausgegeben, in denen in der Praxis im Laufe mehrerer Monate ausgewertete Queries verzeichnet sind[2].

Ein solches Log ist auch für die folgende Evaluation gegeben. Die Queries, die für die Evaluierung letztlich verwendet werden, sollten auch eine realistische Verteilung bezüglich ihrer Größe und Struktur aufweisen. Da die Log-Aufzeichnungen selbst aus dem praktischen Einsatz von DBPedia stammen, ist dies hier bereits gegeben. Die Queries, die in dem Log beinhaltet sind, liegen in URL-kodierter Form vor. In einer vorbereitenden Aufbereitung, werden all diejenigen Queries dekodiert, extrahiert und separat gespeichert, die auch für die vorgesehene Anwendung geeignet sind. Dafür muss eine Query die folgenden Eigenschaften besitzen: (1) sie muss übersetzbar sein, (2) sie darf nur Joins und Triple-Pattern enthalten (andere Ausdrucksmittel werden vom Testsystem bisher nicht unterstützt) und (3) sie muss an dem eingesetzten Datenbestand, d.h. dem DBPedia-Ausschnitt, auszuwerten sein. Letzterer Punkt lässt sich anhand der optimalen Datenstatistik des Datenbestandes beantworten. Die Queries werden zudem nach der Zahl der beinhalteten Joins gruppiert gespeichert.

Aus dem gegebenen Log konnten auf diese Weise insgesamt 86638 Queries extrahieren, die sich bezüglich ihrer Join-Zahl so verteilen, wie in Abbildung 6.1 ersichtlich ist. Eine empirische Untersuchung von Auswertungen realistischer SPARQL-Queries über (u.a.) den DBPedia-Datensatz wird in [AFMPdlF11] vorgenommen und basiert ebenfalls auf der Analyse der an-

[1] Eine aktuelle Aufstellung der in [HMZ10] verwendeten Daten findet sich unter `http://code.google.com/p/fbench/wiki/Datasets`

[2] siehe `http://data.semanticweb.org/usewod/2011/challenge.html`

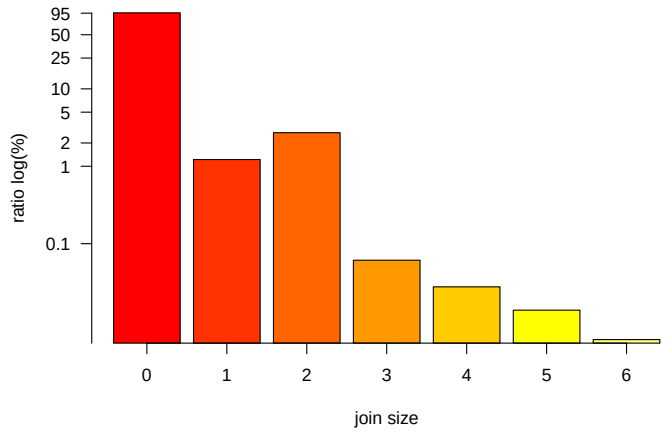


Abbildung 6.1: Die vorgefundene Häufigkeitsverteilung der DBPedia-Queries bezüglich der Zahl der Join-Operatoren.

gesprochenen Logs. Darin wird eine Häufigkeitsverteilung präsentiert, die abgesehen von dem hohen Anteil von Queries der Join-Zahl 0 mit der aus 6.1 übereinstimmt. Das heißt, von der Filterung nach den obigen 3 Bedingungen waren Queries aller Join-Größen offenbar gleichermaßen betroffen und die zur Verfügung stehenden Queries sind realistisch verteilt.

Für die Erzeugung von Reihen k der auszuwertenden Queries, werden aus jeder der Query-Gruppen gemäß einer explizit angegeben Häufigkleitsverteilung anteilsmäßig entsprechend viele Queries ausgewählt und eingebracht. Dabei findet zudem eine zufällige Permutation der Query-Reihe statt. Die Häufigkeitsverteilung wird manuell vorgegeben, orientiert sich dabei aber streng an der aus Abbildung 6.1.

Um den Begriff des „Maßes der Verfälschung“ einer Datenstatistik, wie in Abschnitt 4.1, auch in der Evaluation zu umgehen, werden im Folgenden Datenstatistiken unterschieden, die je jeweils über einem leicht variierten Datenbestand erhoben werden. Die Datenbestände gehen dabei auf eine bekannte Art und Weise durch sukzessive Einbringung randomisierter Veränderungen auseinander hervor. Ausgehend von den Originaldaten wird so eine Transaktionskette, wie sie in 4.1 diskutiert wird, nachgeahmt. Mehr dazu weiter unten in 6.4.

6.3 Untersuchung auf Plausibilität

In diesem Abschnitt werden bereits erste Ergebnisse präsentiert, die durch den Einsatz des beschriebenen Testsystems gewonnen wurden. Ziel ist es hier zunächst, die allgemeine Plausibilität der in Abschnitt 4.2 gezeigten Verfahren zu untersuchen. Für diese Untersuchung wurden mit der im vorherigen Abschnitt beschriebenen Datengrundlage drei separate Query-Auswertungsdurchläufe abgearbeitet. In jedem dieser Durchläufe wird dieselbe vorbereitete Reihe von 1000 zusammengestellten Queries am Testsystem mit dem DBPedia-Ausschnitt ausgewertet und anfallende Beobachtungen aufgezeichnet. Jedoch unterscheiden sich die Durchläufe dabei in der Wahl der verwendeten Datenstatistik, die der Abschätzung der Kardinalitäten der Query-Ergebnisse jeweils zugrunde gelegt wird. Betrachtet werden die Originaldatenstatistik (bezeichnet als $dbpstat_{100\%}$) des vollständigen DBPedia-Datasets sowie zwei Datenstatistiken ($dbpstat_{50\%}$ und $dbpstat_{10\%}$), die je über reduzierte Teilmengen dieses Datasets erhoben wurden. Erzeugt wurden zu diesem Zweck zwei Teilgraphen, die durch Entfernen von Anteilen (50% für Durchgang 2 und 90% für Durchgang 3) der ursprünglichen Tripelmenge gebildet werden. Die zu löschenden Tripel wurden zusätzlich zufällig ausgewählt. Da bei allen der drei Testdurchläufe der tatsächlich abgefragte Datenbestand unverändert bleibt, muss bei Verwendung der beiden dann verfälschten Datenstatistiken $dbpstat_{50\%}$ und $dbpstat_{10\%}$ nach 4.1, gegenüber dem Fall der Originalstatistik, mit allgemein erhöhten Schätzfehlern zu rechnen sein. Auch diese Annahme soll hier untersucht werden.

Für jeden einzelnen Durchlauf wurden nun die bei den Kardinalitätsschätzungen der einzelnen Queries auftretenden (relativen) Fehler beobachtet und aufgezeichnet. Es sei daran erinnert, dass die gewonnen Fehlersequenzen nach der einfachen Filtermethode der Indikatorverfahren von Ausreißern bereinigt wurden (angenommen wurden nur Fehlerwerte e_i mit $1 < e_i \leq 2500$). Im Einzelnen konnten folgenden Werte ermittelt werden:

	$dbpstat_{100\%}$	$dbpstat_{50\%}$	$dbpstat_{10\%}$
Sample-Größe (n)	92	100	107
Durchschnitt ($\bar{e}$)	63.320	83.627	89.646
Standardabw. (s)	198.24	276.80	271.32

Bereits aufgrund dieser Werte lassen sich folgende Schlüsse ziehen:

1. Der durchschnittliche relative Fehler der Kardinalitätsschätzung nimmt allgemein mit der Verfälschung der zugrunde liegenden Datenstatistik

zu.

2. Auch die Varianz der beobachteten Fehler nimmt mit der Verfälschung zu. Insgesamt ergibt sich eine deutliche Veränderung der Fehlerverteilung.

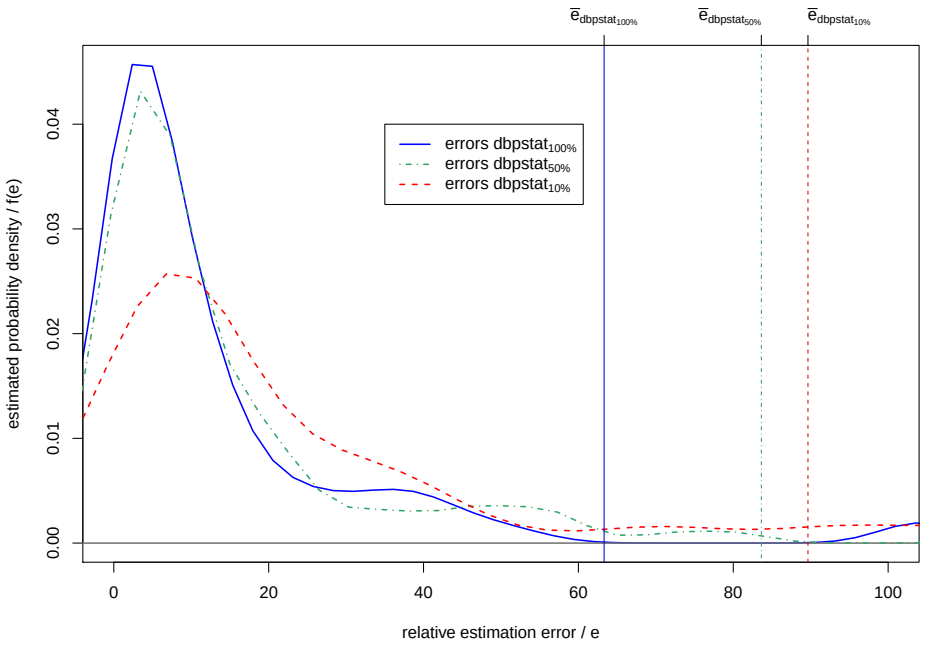


Abbildung 6.2: Geschätzte Wahrscheinlichkeitsdichten der drei unabhängigen Query-Fehlersamples.

Diese Schlussfolgerungen werden durch die Darstellungen in den Abbildungen 6.2 und 6.3 bekräftigt. Abbildung 6.2 beinhaltet die Wahrscheinlichkeitsdichten, die aus den drei gewonnenen Fehlersequenzen abgeschätzt werden können. Hier zeigt sich zum einen visuell die Lageverschiebung der Fehlerverteilung, die mit zunehmender Verfälschung eintritt. Mit der ersten Feststellung verträglich ist die prägnante „Aufspreizung“ der drei Linien für die Fehlerdurchschnitte, die hier mit $\bar{e}_{dbpstat_{100\%}}$, $\bar{e}_{dbpstat_{50\%}}$ und $\bar{e}_{dbpstat_{10\%}}$ bezeichneten sind. Für zunehmend verfälschte Datenstatistiken scheinen sich die Fehlerdurchschnitte nach oben bzw. „rechts“ zu verschieben. Auch die Lage des globalen Maximums der Dichten verschiebt sich dabei zumindest leicht nach oben. Zum anderen ist hier auch die allgmeine Änderung der Fehlerverteilung gemäß Feststellung 2. gut einzusehen. Die Verteilung flacht durch die bereits festgestellte Erhöhung der Varianz offensichtlich ab. Die Unterschiede der Verteilungsdichtefunktionen wirken sich, insbesondere durch

deren Lageverschiebung, auch auf die kumulierten Verteilungsfunktionen der beobachteten Fehler aus. Dies lässt sich an den in Abbildung 6.3 dargestellten Kurven gut beobachten. Tendenziell scheinen die Funktionswerte der Verteilungsfunktionen bei unverfälschten Datenstatistiken größer zu sein, als die jener Verteilungsfunktionen, die aus Fehlerwerten auf Basis verfälschter Datenstatistiken abgeleitet werden können.

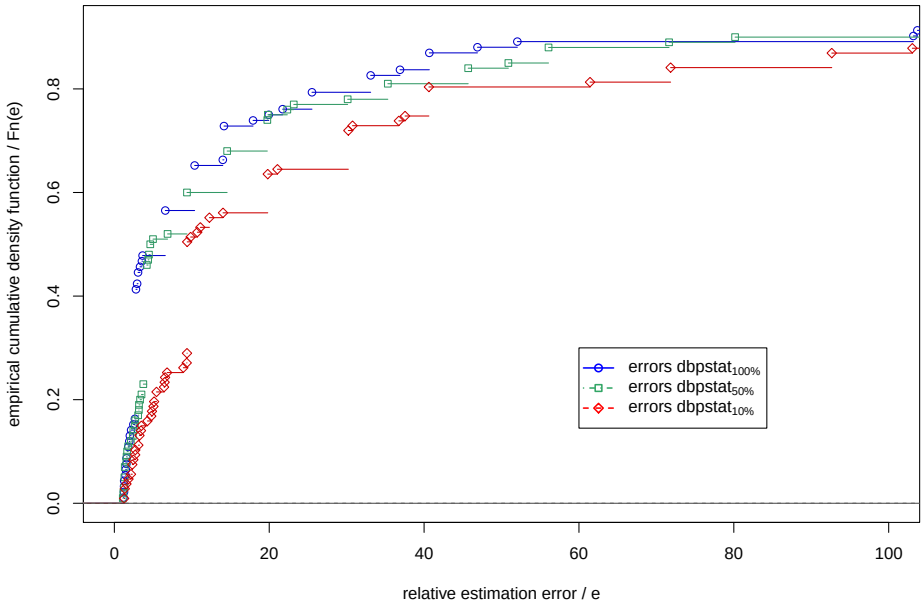


Abbildung 6.3: Empirische (kumulative) Verteilungsunktionen, entsprechend der in 6.2 dargestellten Fehlerdichten.

Die Plausibilität der in Abschnitt 4.2 vorgestellten Verfahren und der grundlegenden Annahmen aus 4.1 ist also insgesamt prinzipiell bestätigt. Der Abgleich aktueller Fehlerwerte mit vorgegebenen absoluten Fehlerobergrenzen (aus 4.2.1) erscheint als zielführend für den Versuch zunehmend verfälschte Datenstatistiken als solche zu erkennen. Dies gilt nach obigen Ergebnissen aber auch für die vergleichende Untersuchungen, die durch die Indikationsverfahren aus 4.2.2 realisiert werden. Für eine weitergehende Beurteilung dieser vergleichenden Verfahren, werden im Folgenden die Ergebnisse, nach Anwendung der ihnen zugrunde liegenden statistischen Tests, auf die drei Fehlersequenzen betrachtet.

p-Werte	$dbpstat_{100\%}$ vs. $dbpstat_{50\%}$	$dbpstat_{100\%}$ vs. $dbpstat_{10\%}$
p_{wt}	0.2787	0.2157
p_{mww}	0.0666	0.0054
p_{ks}	0.0004	$8.902 \cdot 10^{-6}$

Zunächst fällt auf, dass der Zwei-Sample-T-Test nach Welch (mit *wt* abgekürzt), der hier nur zum Vergleich angewendet wurde, in beiden betrachteten Fällen mit 0.2787 und 0.2157 vergleichsweise große p-Werte liefert. Auf Basis nur dieser Werte ließe sich also keinesfalls eine Lageverschiebung der Fehlerwahrscheinlichkeiten für die beiden Fehlersequenzen erkennen. Die bereits in Abschnitt 4.2 begründete Ablehnung dieses Tests erscheint hier als sinnvoll. Die beiden anderen tatsächlich implementierten Tests liefern jedoch zumindest in dem Fall "$dbpstat_{100\%}$ vs. $dbpstat_{10\%}$" p-Werte, die mit vertretbarem α (üblich sind meist 0.05 oder 0.01) eine signifikante Fehlerveränderung, und damit die Obsoleszenz von $dbpstat_{10\%}$, nahelegen. Demgegenüber ließe sich bei „$dbpstat_{100\%}$ vs. $dbpstat_{50\%}$" durch den MWW-Test (im Unterschied zum KS-Test) eine Obsoleszenz von $dbpstat_{10\%}$ noch nicht erkennen. Ohnehin erscheint der KS-Test sensibler auf Veränderungen bei den Fehlerwerten zu reagieren.

Die drei zuvor erzeugten Datenstatistiken $dbpstat_{100\%}$, $dbpstat_{50\%}$ und $dbpstat_{10\%}$ basieren auf Datenbeständen, die zwar aus derselben Datenquelle stammen, sich im Umfang jedoch erheblich unterscheiden. Die gerade diskutierten Ergebnisse deuten daraufhin, dass der Lösungsansatz aus Kapitel 4 grundsätzlich begründet ist und die entwickelten Verfahren für die Erkennung von Verfälschungen des gerade betrachteten Ausmaßes einsetzbar sind. Mehr ist jedoch noch nicht zu sagen. Für die Beurteilung der praktischen Funktionstüchtigkeit der Verfahren müssen auch Verfälschungen erkannt werden können, die durch Veränderungen von Datenbeständen zustande kommen, die: sich 1. über einen größeren Zeitraum erstrecken, 2. durch viele kleinere Schreib-Transaktionen kumulieren und 3.einzeln einen sehr viel kleineren Umfang haben, als gerade betrachtet. Zuletzt wird im nächsten Abschnitt eine erste Untersuchung unter diesen Gesichtspunkten versucht.

6.4 Praktikabilitätstest

Um die in Kapitel 5.2 vorgestellte Systemerweiterung nach den obigen Fragestellungen 3. und 4. auf dessen Praktikabilität hin zu überprüfen, wird im Folgenden die Simulation eines dynamischen Einsatzszenarios mit verän-

derlichen Daten vorgeschlagen und unternommen. Die Verfahren aus 4 zur Erkennung obsoleter Datenstatistiken müssen an einem solchen Einsatzszenario getestet werden.

Im realistischen Umfeld des RDF-Federators treten bei einzelnen Datenquellen Veränderungen an deren Datenbeständen auf, die mit einer ja gerade zu erkennenden Veralterung bzw. Verfälschung der aktuell in Betracht gezogenen Datenstatistiken einhergehen. Nach der Sicht aus 4.1 lässt sich das durch Folgen schreibender Transaktionen modellieren, die auf einen Datenbestand angewendet werden. Jede der Transaktionen bringt Änderungen eines bestimmten Umfangs mit sich, während die Datenstatistik, die für die Datenquelle im Federator vorgehalten wird, jedoch unverändert bleibt. Um in einer Evaluation nun zu zeigen, ob und in welchem Umfang die Verfahren in einer solchen Situation ihre Funktion erfüllen, müssten solche Transaktionsfolgen abgebildet und auf einen realistischen Datenbestand angewendet werden. Dies ist entsprechend sehr aufwendig. Für die weiteren Tests wurde stattdessen ein vereinfachtes Szenario vorbereitet, das im Folgenden beschrieben wird.

Ausgehend von dem bereits angesprochenen DBPedia-Datensatz wurden 20 abgeänderte Teildatensätze abgeleitet. Die Menge dieser Datensätze ist eine Folge, in der jeder der Datensätze aus dem jeweils vorhergehenden durch Entfernen einer bestimmten Anzahl zufällig ausgewählter RDF-Tripel entsteht. Für jeden der dann 21 DBPedia-Datensätze wurde eine zugehörige Datenstatistik (hier in der Form d_i abgekürzt) errechnet und in einer VoID-Beschreibung festgehalten. Aus praktischen Gründen sind es jedoch nicht die Datensätze selbst, die im Testlauf ausgetauscht werden, um eine Transaktionsfolge zu simulieren. Vielmehr werden die erzeugten Datenstatistiken d_0,d_1,...,d_{20} (in dieser Abfolge) nacheinander eingesetzt, während der ursprüngliche Datensatz unverändert verbleibt. Der Austausch des kompletten Datensatzes oder die parallele Verwaltung aller Datensätze wäre zu aufwendig. Hierzu ausgenutzt wird die Definition des relativen Fehlers nach Gl. 4.2b, nach der dieser vom Verhältnis $r_{src,i}$ abhängt. Vereinfachend wird nun $r_{src,i} = \frac{N_0}{N_i}$ angesetzt, mit dem Tripel-Count N_0 der ursprünglichen Datenbestands und dem Tripel-Count N_i eines der 20 abgeleiteten Datensätze. Es wird also unterstellt, dass die konkreten $r_{src,i}$ weitestgehend dem Größenverhältnis von abgeleitetem Datensatz und Ursprungsdatensatz folgen.

Der Umfang der in jedem der Schritte entfernten Tripelmenge wurde nun folgendermaßen bestimmt (eine quantitative Darstellung ergibt sich in Abbildung 6.4):

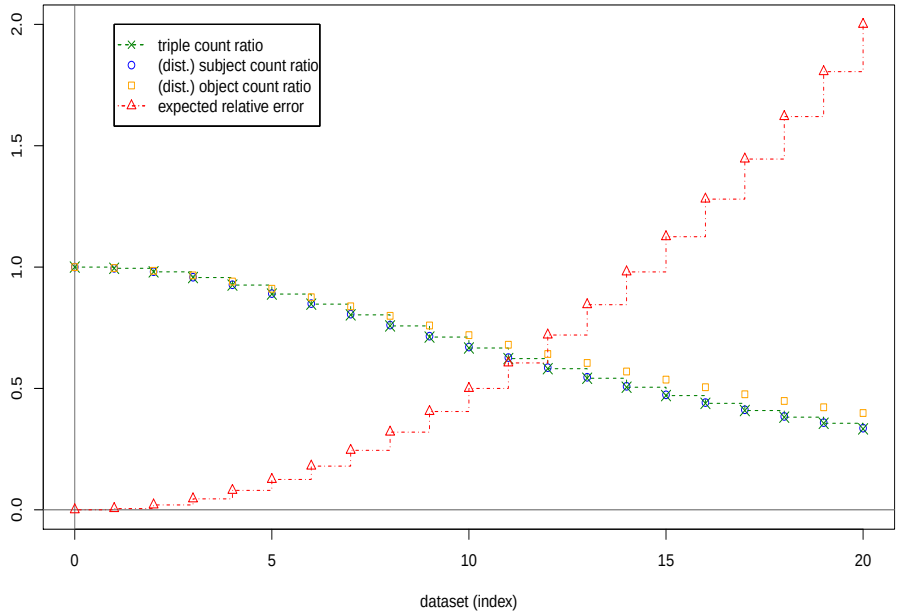


Abbildung 6.4: Quantitative Beschreibung der konstruierten DBPedia-Ausschnitte. Jeder der 21 Datensätze geht durch sukzessives und randomisiertes Entfernen von Tripeln aus dem jeweils vorausgehenden Datensatz hervor. Orientiert am absoluten Triple-Count, wurden die Datensätze so erzeugt, dass spätere Schätzungen über den zugehörigen Datenstatistiken bei Beibehaltung des ursprünglichen Datensatzes Fehler verursachen würden, die nach der dargestellten Kurve quadratisch zunehmen.

- Der Fehler e soll sich im Laufe der simulierten Transaktionsfolge von (dem theoretisch angenommen Wert) 0 bis 2 fortentwickeln, dabei einer Parabel folgen (also $e_i = c \cdot i^2$).
- Die Tripel-Counts N_i entwickeln sich dazu dann mit $N_i = N_0/(c \cdot i^2 + 1)$ monoton nach unten.
- Die resultierenden Verhältnisse $r_{src,i}$ ergeben sich (nach obigen Rechenansatz) genau so, wie sie bei fixer Datenstatistik und einer gleichzeitig realisierten datenvermehrenden Transaktionsfolge (an deren Ende ein verdreifachter Tripel-Count stünde) beobachtbar wären.

Neben den 21 Datenstatistiken, werden zudem 21 separate zufällige Query-Folgen bestehend aus 1000 Queries aus dem im vorherigen Abschnitt erwähnten Query-Log erzeugt. Diese werden in den nachfolgend beschriebenen Testdurchläufen nacheinander ausgeschöpft, die einzelnen Queries jeweils über die gerade aktuelle Datenstatistik d_i abgeschätzt, optimiert und schließlich ausgeführt. Die durch das betriebene Observation-Framework parallel beobachteten Daten dienen der weiteren Analyse. Die möglicherweise eintretenden Indikationsmeldungen werden separat aufgezeichnet und ebenfalls in die Analyse mit aufgenommen.

6.4.1 Testdurchlauf ohne Update der Datenstatistiken

Bei den nun folgend beschriebenen Testdurchläufen werden die erzeugten Datenstatistiken in ihrer Reihenfolge nacheinander in den RDF-Federator eingeladen. Für jede auf diese Weise gerade geladene Datenstatistik wird gleichzeitig eine der Query-Folgen ausgeschöpft, deren Queries jeweils über die aktuelle Datenstatistik abgeschätzt, optimiert und schließlich ausgeführt.

Die aufgezeichneten Daten, die während dieses konkreten Ablaufs gewonnen werden konnten, sind in Abbildung 6.5 einzusehen. Die folgenden Beobachtungen konnten dabei gemacht werden:

1. Mit zunehmender Verfälschung der (aktuellen) Datenstatistik steigt sowohl das Niveau des relativen Fehlers selbst, als auch dessen Varianz an. Die Erkenntnisse aus dem vorherigen Abschnitt werden hier nochmal bekräftigt.
2. Der Verlauf deckt sich in seiner Form überraschend schlecht mit der theoretisch Erwartung, nach der die Erzeugung der Datenstatistiken

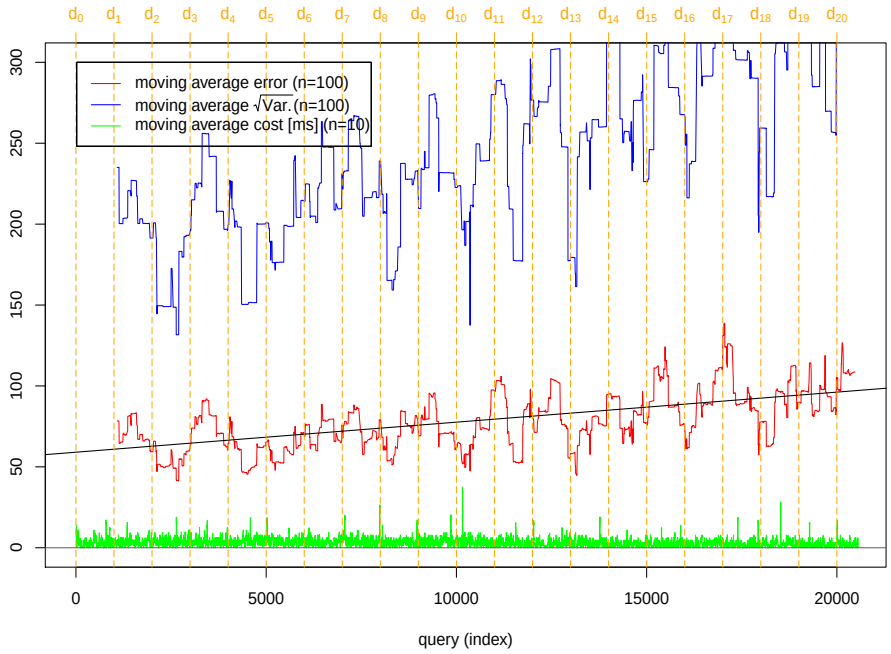


Abbildung 6.5: Gleitende Durchschnitte der tatsächlich beobachteten relativen Fehler (rot) und der Fehlervarianz (blau). Die Breite der gleitenden Datenfenster ist in beiden Fällen n=150. Eingerechnet wurden nur Fehlerwerte e_i mit $1 < e_i \leq 2500$ - dieselbe Filterung wie sie hier auch von den Indikatoren vorgenommen wurde. Zusätzlich eingetragen sind die gleitenden Durchschnitte der von Ausreißern ($>= 5000ms$) bereinigten Auswertungskosten der Queries in ms (grün).

geplant wurde. Die Fehlerkurve lässt sich, wie dargestellt, durch eine einfache Trendgerade mit vergleichsweise geringer Steigung annähern.

3. Der relative Schätzfehler unterscheidet sich bereits zu Beginn der Sequenz deutlich von 0. Dies ist ein starker Hinweis darauf, dass systemische, von den Datenstatistiken unabhängige Anteile am Schätzfehler einen nicht zu unterschätzenden Einfluss haben.

4. Die beobachteten (zeitlichen) Kostenwerte der Queries fallen insgesamt ernüchternd aus. Im Unterschied zu Fehlerwert und -Varianz ist hier keine Abhängigkeit vom Verfälschungsgrad der Datenstatistiken ersichtlich. Dies ist bereits ein deutliches Anzeichen, dass die schlechter werdenden Datenstatistiken zumindest im gegebenen Szenario keine (kostenrelevanten) „Fehloptimierungen“ einzelner Queries nach sich ziehen.

Um den letzten Punkt 4. zu bestätigen, wurde der Testlauf unter expliziter Beobachtung der generierten Pläne zwei weitere Male wiederholt. Einmal mit durchgängiger Beibehaltung der ursprünglichen (d.h. der optimalen) Datenstatistik und einmal mit dem obigen Durchlauf mit zunehmend schlechten Datenstatistiken (d_0,d_1,..,d_{20}). Tatsächlich ergaben sich dabei in keinem einzigen Fall für eine Query zwei unterschiedliche Pläne. Das bedeutete auch, dass sich an dem gegebenen auf Praxisnähe bedachten Szenario etwaige Kostenvorteile durch den Einsatz der entwickelten Obsoleszenz-Indikatoren ausschlossen. Eine explizite vergleichende Untersuchung erübrigte sich dadurch und wurde nicht weiter verfolgt.

6.4.2 Testdurchläufe mit Updates nach je einem Indikator

Der gerade beschrieben Testdurchlauf wurde nun so abgeändert, dass eintreffende Indikationen das sofortige Laden der optimalen bzw. ursprünglichen Datenstatistik d_0 verursachen. Die Abfolge der Datenstatistiken, die nach Abschluss einer der 21 Query-Folgen wie oben weiterbetrieben wird, beginnt dann von vorne. Zudem wird je Durchlauf nur eine Indicator-Instanz betrieben. Im Folgenden werden die Beobachtungen während dreier verschiedener solcher Testdurchläufe aufgeführt. Diese unterscheiden sich lediglich im jeweils betriebenen Indikator:

1. Indicator-Instanz: ErrorTresholdIndicator (siehe Abbildung 6.6)
    - Fensterweite: 100 Datenpunkte

- Fehler-Treshold: 100

2. Indicator-Instanz: MWWIndicator (siehe Abbildung 6.7)
   - Fensterweite: 100 Datenpunkte
   - Fensteranzahl: 3
   - α: 0.1
3. Indicator-Instanz: KSIndicator (siehe Abbildung 6.8)
   - Fensterweite: 100 Datenpunkte
   - Fensteranzahl: 3
   - α: 0.1

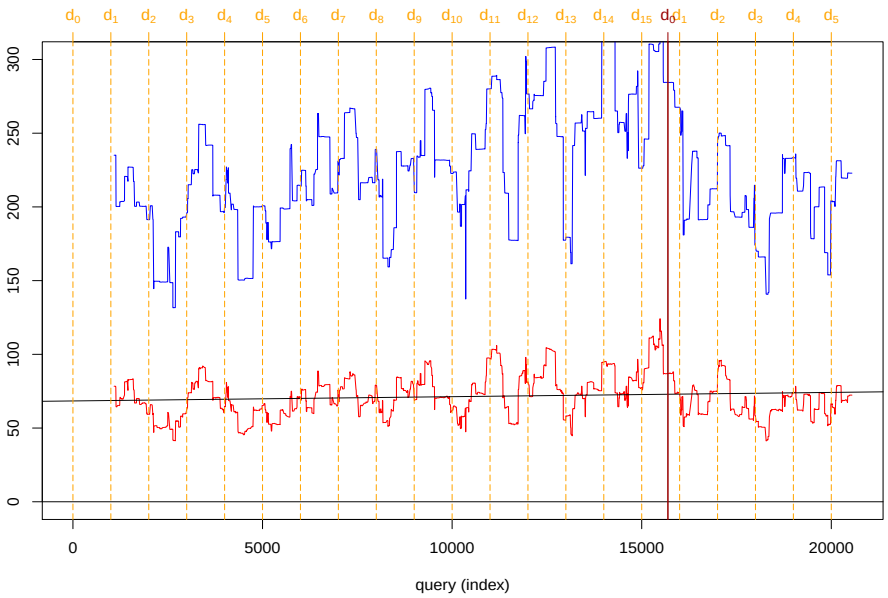


Abbildung 6.6

Die Beobachtungen, die während der drei Testdurchläufe gemacht werden konnten, zeigen zum Teil deutliche Unterschiede zwischen den jeweils betrachteten Indikationsverfahren (in Abhängigkeit von den festgelegten Parametern) auf.

Während des gesamten Testdurchlauf 1 kommt es nur zu einer einzigen Indikationsmeldung durch die eingesetzte ErrorThresholdIndicator-Instanz. Erst nach Verarbeitung der Query an Stelle 15691 überschritt der über dem Datenfenster mit Weite 100 erhobene Fehlerdurchschnitt den angegebenen

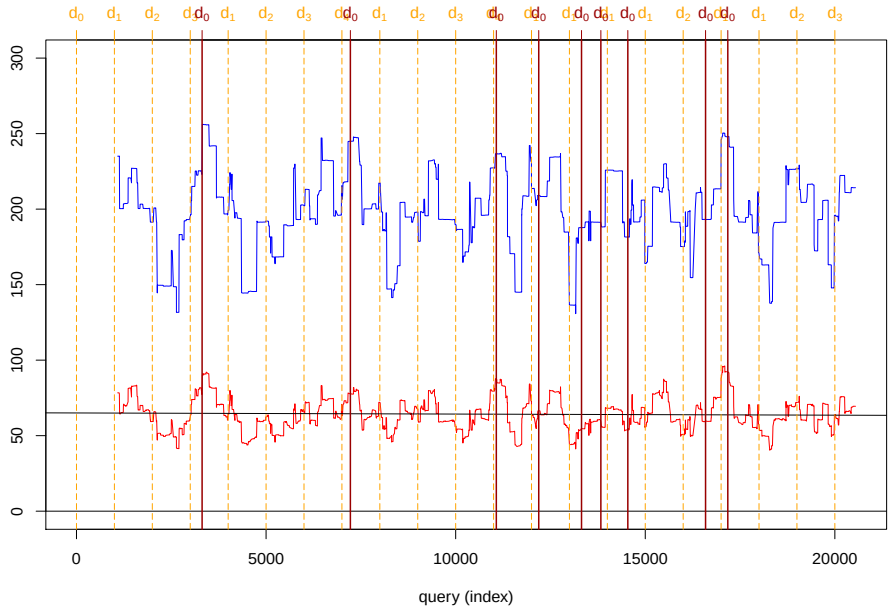


Abbildung 6.7

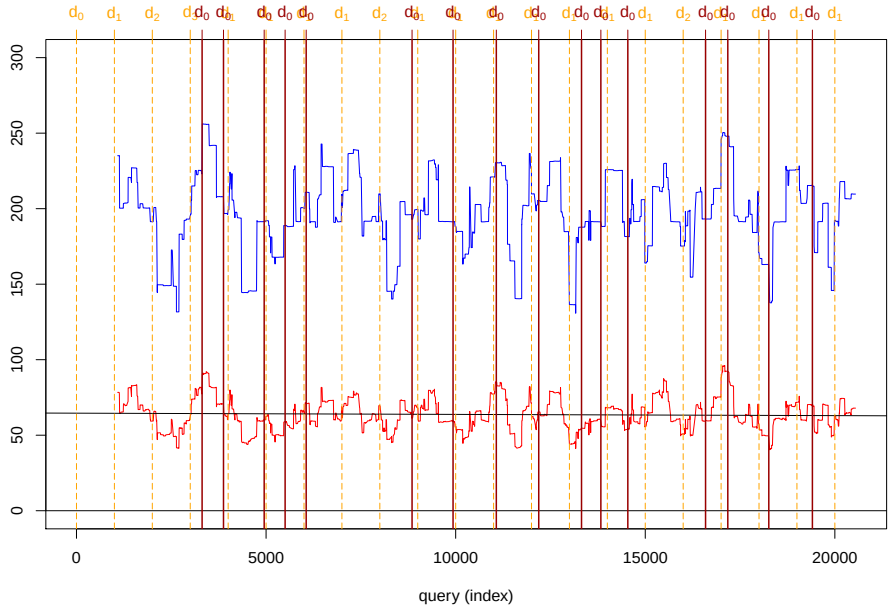


Abbildung 6.8

Schwellwert von 100. Anschließend begann die Abfolge der Datenstatistiken wieder bei der optimal angepassten Datenstatik d_0 und während der übrigen Queries blieb der Fehlerdurchschnitt unter dieser Obergrenze. Deutlich zeigt sich dies in der in Abbildung 6.6 dargestellten Fehlerkurve, die sich bis zu dieser Stelle mit der Fehlerkurve aus Abbildung 6.5 deckt. Durch die angestoßene Rücksetzung auf die optimale Datenstatistik, sinkt demgegenüber hier der Fehler danach jedoch wieder, sodass sich für die gesamte Fehlerkurve ein deutlich flacherer Trend annähern lässt.

Bei Testdurchlauf 2 zeigt der hier eingesetzte Indikator (MWWIndicator) bereits eine höhere Sensitivität. Insgesamt kam es hier zu 9 Indikationsmeldungen, die über einen großen Bereich der der Sequenz verteilt sind. Das liegt auch darin begründet, dass es sich hier um ein Verfahren handelt, bei dem angrenzende Teilsequenzen verglichen werden. Da der Fehlertrend (im Fall ohne Update), wie gesehen, eine Gerade positiver Steigung nachvollzieht, sind Indikationen hier zu Beginn der Fehlersequenz ebenso gut möglich, wie zu deren Ende hin. Durch die häufigen Indikationsmeldungen und die jeweils eintretenden Zurücksetzungen der Datenstatistik, weist der Trend der resultierenden Fehlerkurve bereits praktisch keine Steigung mehr auf.

Ein ähnliches Bild ergibt sich schließlich in Testdurchlauf 3, in dem eine Instanz des implementierten KSIndicators betrieben wird. Der Indikator, der wie beschrieben, auf allgemeine Unterschiede der kumulierten Fehlerverteilung zweier Fehler(teil)sequenzen prüft, löste noch häufiger Meldungen aus. Insgesamt kam es zu 16 Indikationsmeldungen, die wie in Durchlauf 2 und aus denselben Gründen, einen großen Bereich der aufgezeichneten Fehlersequenz überspannen. Auch hier resultieren die häufigen Meldungen letztlich in einer Fehlersequenz mit nahezu horizontalem Trend.

6.5 Diskussion und Lessons Learned

Zunächst kann noch einmal festgehalten werden, dass die entwickelten Verfahren (siehe 4.2) prinzipiell auf das gestellte Problem anwendbar sind. Die Fehlerwerte entwickeln sich bei zunehmender Verfälschung einer Datenstatistik in Übereinstimmung mit der Vorstellung aus 4.1. Sowohl die Ergebnisse der Plausibilitätsuntersuchung, als auch die gewonnen Daten aus den Praktikabilitätstests zeigen das an. Gerade die Praktikabilitätstests haben jedoch auch gezeigt, dass der Gesamtansatz, der der präsentierten Entwicklung zugrundliegt, letztlich trotzdem in Zweifel zu ziehen ist.

Nach Ansicht des Autors tragen die folgenden Punkte zu dieser Feststellung bei:

- Die Fehlerkurve lässt sich durch den Einsatz der Indikatoren und ggf. ausgelöste Updates der Datenstatistik abflachen. Dies wirkt sich jedoch mitnichten auf die tatsächlich stattfindenden Optimierungsvorgänge aus, sodass keinerlei Kostenersparnis bei der Query-Auswertung erwächst.
- Der Kardinalitätsschätzer besitzt offenbar eine nicht zu unterschätzende Ungenauigkeit, die unabhängig von eingegeben Datenstatistiken zu Fehlern führt, die sich mit der Join-Größe (nach [IC91]) exponentiell entwickeln.
- Der Austausch veralteter Datenstatistiken beeinflusst das Kostenmodell vermutlich nicht so direkt, wie andere kostenrelevante Einflüsse. Also solche wie z.B. die aktuelle technische Leistungsfähigkeit am Gesamtsystem beteiligter Teilkomponenten (z.B. die Auslastung der Server angeschlossener Datenquellen).
- Signifikante Veränderungen (gemäß der Indikatoren) am Datenbestand einer föderierten Datenquelle, stellen sich wahrscheinlich in der Praxis erst über Zeitspannen ein, die noch sehr viel größer sind, als hier betrachtet. Viel dynamischere Einflüsse auf die Kostenschätzung werden hingegen gar nicht beobachtet oder untersucht.

Insgesamt lohnt sich der Einsatz der Systemerweiterung somit nicht. Zum einen vereinfacht sie die gegebene Situation zu stark, zum anderen untersucht sie einen Kosteneinfluss, der durch dynamischere Faktoren in der Praxis unter Umständen leicht übertroffen werden kann. Gleichwohl lohnt sich die Beobachtung und Dokumentation von Messwerten aus der Query-Verarbeitung, zumal sich die grundlegende Überlegung aus 4.1 zu bestätigen scheint. Jedoch sollte eine Neuerhebung einzelner Datenstatistiken durch einen Systembetreuer vorgenommen werden. Diesem könnte dazu eine visuelle Aufbereitung der Beobachtungen und Analyseergebnisse, wie sie hier auftreten nach Bedarf präsentiert werden.

Trotz der gerade getroffenen Schlussfolgerungen lassen sich dennoch einige Verbesserungen bzw. Weiterentwicklungen der dargestellten Lösung diskutieren. Die in den letzten beiden dargestellten Testdurchläufen verwendeten Indikatoren, zeichnen sich jeweils durch eine hohe Sensibilität gegenüber den zu beobachtenden Schätzfehlern aus. Dementsprechend werden in beiden Fällen viele Indikationsmeldungen gemacht. In den durchgeführten Tests wurde die Neuerhebung der Datenstatistik durch Zurücksetzen auf den An-

fang der bereits vorbereiteten Folge von Datenstatistiken simuliert. Eine tatsächliche Neuerhebung eines Datensatzes vom Umfang des hier verwendeten DBPedia-Ausschnitts wäre jedoch sehr langwierig (2-3 Stunden bei dem hier verwendeten Rechner). Für besonders sensible Verfahren sollte nicht nach jeder Indikationsmeldung solche Neuerhebungen auch direkt stattfinden. Eine „Polling"-Strategie, in der z.B. für feste Zeiträume eine Mindestzahl von Indikationsmeldungen eintreffen muss, um dann tatsächlich eine Neuerhebung der betroffenen Datenstatistik auszulösen. Auch kann man eine kombinierte Analyse der Fehlersequenzen in Betracht ziehen, in der die Indikationsmeldungen mehrerer Indikatoren nach einer bestimmten Strategie miteinander zu einem Gesamturteil zusammengeführt werden. Dies würde eine Abhängigkeit von besonders sensiblen Indikatoren vermindern, zumal die hier Betrachteten Verfahren sich bereits darin elementar darin unterscheiden, dass eines der drei Verfahren (Verfahren 1 / ErrorTresholdIndicator) absolut ist, während die anderen beiden vergleichende bzw. in Relation setzende Verfahren sind.

Eine wirklich vielversprechende Weiterverwendung der präsentierten Entwicklung bestünde jedoch, wie angedeutet, vermutlich nur in adaptiven Systemen, die andere Kosteneinflüsse zur Beobachtungsgrundlage haben. Verwandte Arbeiten in dem Zusammenhang wurden bereits in den Abschnitten 3.2 und 3.3 angesprochen.

7 Fazit

Im Rahmen der vorliegenden Arbeit wurde für ein existierendes föderiertes Datenbanksystem eine Erweiterung entwickelt, die die Kardinalitätsschätzung von Queries auf der Basis gegebener statistischer Abbilder (Datenstatistiken) der verteilten Datenquellen automatisch analysieren soll. Aufgabe der Erweiterung ist schließlich, die beteiligten Datenstatistiken neu zu erheben, wenn während der fortgesetzten Untersuchung verschiedene Indikatoren eine Verschlechterung der Schätzung anzeigen, als deren Ursache verfälschte oder veraltete Datenstatistiken angesehen werden. Es wurde demonstriert, wie dies anhand der Beobachtung und parallel ablaufenden Analyse anfallender Fehlerwerte unternommen werden kann.

Die notwendige Entwicklung eines Rahmenwerks (Observation-Framework), das diese Beobachtung im laufenden Betrieb des Datenbanksystems erst ermöglicht, wurde ausführlich dokumentiert. Ein Datenmodell, das sich an der Struktur eingehender Abfragen orientiert und aus der Abfrageoptimierung und Query-Feedback gewonnene Messwerte aufnehmen kann, wurde spezifiziert. Das Verhalten der einzelnen aktiven Komponenten des Observation-Frameworks und deren Zusammenspiel mit dem bestehenden Datenbanksystem wurden ebenso beschrieben. Als Indikatoren, die die Obsoleszenz einzelner Datenstatistiken untersuchen sollen, wurden drei verschiedene Verfahren vorgeschlagen, die auf mehreren vom Observation-Framework fortgeschriebenen Fehlersequenzen als Eingabe arbeiten. Die den Verfahren zugrundeliegenden Annahmen und Überlegungen wurden dabei ebenso dargestellt, wie ihre Funktionsweise.

In Rahmen einer Evaluation wurde eine Testumgebung erzeugt, deren Entwurf genau dargelegt wurde. Es wurde zunächst untersucht, inwiefern die Annahmen, die den Indikator-Verfahren zugrunde liegen, in der Praxis zutreffend sind. Es konnte hier gezeigt werden, dass diese plausibel sind und die Indikatoren damit prinzipiell für ihren Zweck geeignet sind. In einer weitergehenden, an der Praxis orientierten Untersuchung, ergaben sich jedoch zum Teil ernüchternde Ergebnisse, die anzweifeln lassen, dass sich mit Hilfe der beschriebenen Systemerweiterung auch zählbare Kostenvorteile ergeben können. Die Gründe dafür wurden diskutiert.

Der Austausch von Datenstatistiken als Stellschraube eines adaptiven Systems wie diesem, kann zumindest in Situationen, wie der untersuchten , insgesamt als unwirksam angesehen werden. Jedoch lohnt es sich den Versuch

zu unternehmen, anderen direkteren Kosteneinflüssen durch Anpassung des Kostenmodells des Federators selbst zu begegnen. Verwandte Arbeiten, die hier erläutert wurden, greifen solche Ideen bereits auf. Zumindest das beschriebene Observation-Framework ließe sich vermutlich dazu wiederverwenden.

Literaturverzeichnis

[ACPS96] S. Adali, K. S. Candan, Y. Papakonstantinou, and V. S. Subrahmanian. Query caching and optimization in distributed mediator systems. In *Proceedings of the 1996 ACM SIGMOD international conference on Management of data*, SIGMOD '96, pages 137–146, New York, NY, USA, 1996. ACM.

[AFMPdlF11] Mario Arias, Javier D. Fernández, Miguel A. Martínez-Prieto, and Pablo de la Fuente. An Empirical Study of Real-World SPARQL Queries. *CoRR*, abs/1103.5043, 2011.

[AH00] Ron Avnur and Joseph M. Hellerstein. Eddies: continuously adaptive query processing. In *Proceedings of the 2000 ACM SIGMOD international conference on Management of data*, SIGMOD '00, pages 261–272, New York, NY, USA, 2000. ACM.

[BB06] David Beckett and Art Barstow. N-Triples. `http://www.w3.org/2001/sw/RDFCore/ntriples/`, 2006. [Online; letzter Zugriff am 22-März-2011].

[BBL11] David Beckett and Tim Berners-Lee. Turtle - Terse RDF Triple Language. `http://www.w3.org/TeamSubmission/turtle/`, 2011. [Online; letzter Zugriff am 22-März-2011].

[BG04] Dan Brickley and R.V. Guha. RDF Vocabulary Description Language 1.0: RDF Schema. `http://www.w3.org/TR/rdf-schema/`, 2004. [Online; letzter Zugriff am 16-Februar-2010].

[BL01] Tim Berners-Lee. Notation 3. `http://www.w3.org/DesignIssues/Notation3.html`, 2001. [Online; letzter Zugriff am 22-März-2011].

[BL08] Tim Berners-Lee. Linked Open Data. `http://www.w3.org/2008/Talks/0617-lod-tbl/`, 2008. [Online; letzter Zugriff am 3-April-2011].

[BLFM05] Tim Berners-Lee, Roy Fielding, and Larry Masinter. Uniform Resource Identifier (URI): Generic Syntax. `http://tools.ietf.org/html/rfc3986`, January 2005. [Online; letzter Zugriff am 22-März-2011].

[BM04] Dave Beckett and Brian McBride. RDF/XML Syntax Specification (Revised). http://www.w3.org/TR/rdf-syntax-grammar/, 2004. [Online; letzter Zugriff am 22-März-2011].

[BPSM+08] Tim Bray, Jean Paoli, C. M. Sperberg-McQueen, Eve Maler, and François Yergeau. Extensible Markup Language (XML) 1.0 (Fifth Edition). http://www.w3.org/TR/2008/REC-xml-20081126/, 2008. [Online; letzter Zugriff am 22-März-2011].

[Bra07] Steve Bratt. Semantic Web, and Other Technologies to Watch. http://www.w3.org/2007/Talks/0130-sb-W3CTechSemWeb, 2007. [Online; letzter Zugriff am 18-März-2011].

[Bro10] Dominik Brosius. Entwicklung eines generischen SesameSails für die Abbildung von SPARQL-Anfragen auf Webservices. http://kola.opus.hbz-nrw.de/volltexte/2010/515/pdf/SA_brosius.pdf, 2010. [Online; letzter Zugriff am 15-Februar-2011].

[Cod70] E. F. Codd. A relational model of data for large shared data banks. *Commun. ACM*, 13:377–387, June 1970.

[Con97] Stefan Conrad. *Föderierte Datenbanksysteme*. Springer, Berlin [u.a.], 1997.

[DIR07] Amol Deshpande, Zachary G. Ives, and Vijayshankar Raman. Adaptive Query Processing. *Foundations and Trends in Databases*, 1(1):1–140, 2007.

[DKS92] Weimin Du, Ravi Krishnamurthy, and Ming-Chien Shan. Query Optimization in a Heterogeneous DBMS. In Li-Yan Yuan, editor, *18th International Conference on Very Large Data Bases, August 23-27, 1992, Vancouver, Canada, Proceedings*, pages 277–291. Morgan Kaufmann, 1992.

[EKMR06a] Stephan Ewen, Holger Kache, Volker Markl, and Vijayshankar Raman. Progressive Query Optimization for Federated Queries. In Yannis Ioannidis, Marc Scholl, Joachim Schmidt, Florian Matthes, Mike Hatzopoulos, Klemens Boehm, Alfons Kemper, Torsten Grust, and Christian Boehm, editors, *Advances in Database Technology - EDBT 2006*, volume 3896 of

Lecture Notes in Computer Science, pages 847–864. Springer Berlin / Heidelberg, 2006.

[EKMR06b] Stephan Ewen, Holger Kache, Volker Markl, and Vijayshankar Raman. Progressive Query Optimization for Federated Queries. In Yannis Ioannidis, Marc Scholl, Joachim Schmidt, Florian Matthes, Mike Hatzopoulos, Klemens Boehm, Alfons Kemper, Torsten Grust, and Christian Boehm, editors, *Advances in Database Technology - EDBT 2006*, volume 3896 of *Lecture Notes in Computer Science*, pages 847–864. Springer Berlin / Heidelberg, 2006.

[FKPT03] Ludwig Fahrmeir, Rita Künstler, Iris Pigeot, and Gerhard Tutz, editors. *Statistik*. Springer-Lehrbuch. Springer, Berlin [u.a.], 4., verb. aufl. edition, 2003.

[GS11] Olaf Görlitz and Steffen Staab. Federated Data Management and Query Optimization for Linked Open Data. In Athena Vakali and Lakhmi Jain, editors, *New Directions in Web Data Management 1*, volume 331 of *Studies in Computational Intelligence*, pages 109–137. Springer Berlin / Heidelberg, 2011.

[GST96] Georges Gardarin, Fei Sha, and Zhao-Hui Tang. Calibrating the Query Optimizer Cost Model of IRO-DB, an Object-Oriented Federated Database System. In T. M. Vijayaraman, Alejandro P. Buchmann, C. Mohan, and Nandlal L. Sarda, editors, *VLDB'96, Proceedings of 22th International Conference on Very Large Data Bases, September 3-6, 1996, Mumbai (Bombay), India*, pages 378–389. Morgan Kaufmann, 1996.

[HBF09] Olaf Hartig, Christian Bizer, and Johann-Christoph Freytag. Executing SPARQL Queries over the Web of Linked Data. In *8th International Semantic Web Conference*, volume 5823 of *Lecture Notes in Computer Science*, pages 293–309. Springer-Verlag, Oktober 2009.

[HHK+10] Andreas Harth, Katja Hose, Marcel Karnstedt, Axel Polleres, Kai-Uwe Sattler, and Jürgen Umbrich. Data summaries for on-demand queries over linked data. In Michael Rappa, Paul Jones, Juliana Freire, and Soumen Chakrabarti, editors, *WWW*, pages 411–420. ACM, April 2010.

[HKRS08] Pascal Hitzler, Markus Krötzsch, Sebastian Rudolph, and

York Sure. *Semantic Web – Grundlagen.* Springer, Berlin [u.a.], 2008.

[HMZ10] Peter Haase, Tobias Mathäß, and Michael Ziller. An evaluation of approaches to federated query processing over linked data. In *Proceedings of the 6th International Conference on Semantic Systems*, I-SEMANTICS '10, pages 5:1–5:9, New York, NY, USA, 2010. ACM.

[HPlG06] Justo Hidalgo, Alberto Pan, Manuel Álvarez, and Jaime Guerrero. Efficiently Updating Cost Repository Values for Query Optimization on Web Data Sources in a Mediator/Wrapper Environment. In Opher Etzion, Tsvi Kuflik, and Amihai Motro, editors, *Next Generation Information Technologies and Systems*, volume 4032 of *Lecture Notes in Computer Science*, pages 1–12. Springer Berlin / Heidelberg, 2006.

[IC91] Yannis E. Ioannidis and Stavros Christodoulakis. On the propagation of errors in the size of join results. *SIGMOD Rec.*, 20:268–277, April 1991.

[IK84] Toshihide Ibaraki and Tiko Kameda. On the optimal nesting order for computing N-relational joins. *ACM Transactions on Database Systems*, 9:482–502, September 1984.

[Ioa96] Yannis E. Ioannidis. Query Optimization. *ACM Computing Surveys (CSUR)*, 28(1):121–123, 1996.

[JK84] Matthias Jarke and Jürgen Koch. Query Optimization in Database Systems. *ACM Computing Surveys (CSUR)*, 16(2):111–152, 1984.

[KD98] Navin Kabra and David J. DeWitt. Efficient mid-query reoptimization of sub-optimal query execution plans. In *Proceedings of the 1998 ACM SIGMOD international conference on Management of data*, SIGMOD '98, pages 106–117, New York, NY, USA, 1998. ACM.

[KE06] Alfons Kemper and André Eickler. *Datenbanksysteme - Eine Einführung, 6. Auflage.* Oldenbourg, 2006. Kapitel 8.

[Krc05] Helmut Krcmar. *Informationsmanagement (4. Aufl.).* Springer, 2005.

[LS99] Ora Lassila and Ralph Swick. Resource Description Framework (RDF) Model and Syntax Specification. `http://www.`

w3.org/TR/1999/REC-rdf-syntax-19990222/, 1999. [Online; letzter Zugriff am 15-Februar-2010].

[LT10] Günter Ladwig and Thanh Tran. Linked Data Query Processing Strategies. In *International Semantic Web Conference*, volume 6496 of *Lecture Notes in Computer Science*, pages 453–469. Springer, November 2010.

[MdS07] Joaquim Marques de Sá. Non-Parametric Tests of Hypotheses. In Joaquim P. Marques de Sá, editor, *Applied Statistics Using SPSS, STATISTICA, MATLAB and R*. Springer Berlin Heidelberg, 2007.

[Mit95] Bernhard Mitschang. *Anfrageverarbeitung in Datenbanksystemen - Entwurfs- und Implementierungskonzepte*. Vieweg, 1995.

[ML86a] Lothar F. Mackert and Guy M. Lohman. R* Optimizer Validation and Performance Evaluation for Distributed Queries. In Wesley W. Chu, Georges Gardarin, Setsuo Ohsuga, and Yahiko Kambayashi, editors, *VLDB'86 Twelfth International Conference on Very Large Data Bases, August 25-28, 1986, Kyoto, Japan, Proceedings*, pages 149–159. Morgan Kaufmann, 1986.

[ML86b] Lothar F. Mackert and Guy M. Lohman. R* optimizer validation and performance evaluation for local queries. In *Proceedings of the 1986 ACM SIGMOD international conference on Management of data*, SIGMOD '86, pages 84–95, New York, NY, USA, 1986. ACM.

[ML89] Lothar F. Mackert and Guy M. Lohman. Index scans using a finite LRU buffer: a validated I/O model. *ACM Trans. Database Syst.*, 14:401–424, September 1989.

[MRS+04] Volker Markl, Vijayshankar Raman, David Simmen, Guy Lohman, Hamid Pirahesh, and Miso Cilimdzic. Robust query processing through progressive optimization. In *Proceedings of the 2004 ACM SIGMOD international conference on Management of data*, SIGMOD '04, pages 659–670, New York, NY, USA, 2004. ACM.

[NGT98] Hubert Naacke, Georges Gardarin, and Anthony Tomasic. Leveraging Mediator Cost Models with Heterogeneous Data Sources. In *Proceedings of the Fourteenth International Con-*

ference on Data Engineering, February 23-27, 1998, Orlando, Florida, USA, pages 351–360. IEEE Computer Society, 1998.

[Pol97] Wolfgang Polasek. *Schließende Statistik*. Springer-Lehrbuch. Springer, Berlin [u.a.], 1997.

[PS08] Eric Prud'hommeaux and Andy Seaborne. SPARQL Query Language for RDF. `http://www.w3.org/TR/rdf-sparql-query/`, 2008. [Online; letzter Zugriff am 15-Februar-2011].

[Rus] Jack Rusher. Triple Store. `http://www.w3.org/2001/sw/Europe/events/20031113-storage/positions/rusher.html`. [Online; letzter Zugriff am 7-April-2011].

[RZL04] Amira Rahal, Qiang Zhu, and Per-Åke Larson. Evolutionary techniques for updating query cost models in a dynamic multidatabase environment. *The VLDB Journal*, 13:162–176, 2004.

[SAC+79] P. Griffiths Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, and T. G. Price. Access path selection in a relational database management system. In *Proceedings of the 1979 ACM SIGMOD international conference on Management of data*, SIGMOD '79, pages 23–34, New York, NY, USA, 1979. ACM.

[SLMK01] Michael Stillger, Guy M. Lohman, Volker Markl, and Mokhtar Kandil. LEO - DB2's LEarning Optimizer. In *VLDB*, pages 19–28, 2001.

[WG09] W3C OWL Working Group. OWL 2 Web Ontology Language. `http://www.w3.org/TR/owl-overview/`, Oktober 2009. [Online; letzter Zugriff am 23-Februar-2011].

[Wie92] Gio Wiederhold. Mediators in the Architecture of Future Information Systems. *IEEE Computer*, 25(3):38–49, 1992.

[ZL94] Qiang Zhu and Per-Åke Larson. A Query Sampling Method of Estimating Local Cost Parameters in a Multidatabase System. In *Proceedings of the Tenth International Conference on Data Engineering, February 14-18, 1994, Houston, Texas, USA*, pages 144–153. IEEE Computer Society, 1994.