Skalierbare & Push-basierte Echtzeitanfragen für Pull-basierte Datenbanken[1]

Wolfram Wingerath[2]

Abstract: Traditionelle Datenbanksysteme sind für Pull-basierte Anfragen optimiert, d.h. sie stellen Informationen als direkte Antwort auf Anfrage eines Klienten zur Verfügung. Dieses Zugriffsmuster ist zwar für überwiegend statische Domänen praktikabel, erfordert allerdings ineffiziente und langsame Workarounds (z.B. periodische Neuauswertung einer Anfrage), wenn die Klienten auf dem neuesten Stand gehalten werden müssen. Moderne Echtzeitdatenbanken beheben diesen Mangel zwar konzeptuell, indem sie Ergebnisaktualisierungen durch Push-basierte Echtzeitanfragen proaktiv an ihre Klienten ausliefern. Die derzeitig auf dem Markt befindlichen Systeme sind jedoch nur von begrenzter praktischer Relevanz, da sie schwer in bestehende Anwendungen zu integrieren sind, mangelhafte Skalierbarkeit aufweisen oder komplexe Anfragen von vornherein nicht unterstützen. Um diese Probleme zu lösen, schlagen wir in dieser Dissertation das Systemdesign InvaliDB vor, welches lineare Lese- und Schreibskalierbarkeit für ausdrucksmächtige Echtzeitanfragen als Opt-in-Feature für Pull-basierte Datenbanksysteme bereitstellt. InvaliDB befindet sich seit Juli 2017 im produktiven Einsatz als Teil der Backend-as-a-Service-Plattform der Firma Baqend.

1 Motivation: Push-basierte Datenbanken braucht das Land

Die hier vorgestellte Arbeit [Wi19] behandelt ein skalierbares Systemdesign, welches traditionelle Pull-basierte Datenbanksysteme um Push-basierte Echtzeitanfragen erweitert.

Heute informieren immer mehr (Web-)Anwendungen ihre Nutzer über Status-Updates und andere Ereignisse in Echtzeit und so gehen Nutzer oft schon davon aus, dass sich die Aktivitäten der anderen im eigenen Benutzungskontext widerspiegeln. Beispiele für derartige Anwendungen sind kollaborative Online-Dokumente und Webseiten, die auf soziale Interaktion abzielen (soziale Medien). Datenzugriff ist hier *Push-basiert*, weil Informationen sofort bei Veröffentlichung proaktiv an den Nutzer herangetragen („gepusht“) werden. Traditionelle **Datenbankmanagementsysteme** (DBMSs oder schlicht Datenbanken) [He07] sind allerdings auf *Pull-basierten* Datenzugriff ausgelegt, bei dem Informationen nur auf Anfrage des Klienten übermittelt werden. Trotz Erweiterung um Trigger und andere Push-orientierte Konzepte werden traditionelle Datenbanken bei Push-basierten Zugriffsmustern von nativ Push-basierten Systemen hinsichtlich der Performance um mehrere Größenordnungen übertroffen [SC05]. Die Unzulänglichkeit traditioneller Datenbanktechnologie für den Umgang mit sich schnell ändernden Daten wird allgemein als eine der grundlegenden Herausforderungen beim Entwurf von Datenbanksystemen angesehen [SCZ05].

Um Aktualisierungen mit geringer Latenz in Domänen mit hoher Änderungsrate zu gewährleisten, brechen **Datenstrommanagementsysteme** (DSMSs) [GZ10] mit der Idee

[1] Englischer Titel der Dissertation: „Scalable Push-Based Real-Time Queries on Top of Pull-Based Databases“

[2] wolle@baqend.com, Baqend GmbH (vorgeschlagen durch die Universität Hamburg)

eines persistent Datenspeichers. Statt beliebiger ad hoc-Anfragen über Collections (Datensammlungen) werten sie sequentielle, lang laufende Anfragen über Datenströme aus. DSMSs erzeugen neue Ausgaben, sobald neue Daten verfügbar werden, und sind daher nativ Push-basiert. Die Daten können jedoch nur in einem einzigen Durchgang verarbeitet werden, da Datenströme konzeptionell *unbegrenzte* Sequenzen von Datenelementen sind und daher nicht für beliebige Zeit aufbewahrt werden können. Datenstromanfragen können daher nur über Daten ausgewertet werden, die nach ihrer Aktivierung eingehen.

	Datenbankmanagement	**Datenstrommanagement**
Datenzugriff	Pull-basiert	Push-basiert
Datenmodell	persistente Collections	vergängliche Datenströme
Anfrageauswertung	ad hoc, random access	kontinuierlich, sequenziell

Tab. 1: Ein direkter Vergleich der Kernmerkmale von Datenbank- und Datenstrommanagement.

Datenbank- und Datenstrommanagement folgen jeweils grundlegend unterschiedlichen Semantiken bei Datenverarbeitung und -zugriff, wie Tabelle 1 zusammenfasst. Datenbankanfragen über persistente Collections sind für Anwendungen ideal, die eine (konsistente) Sicht auf ihre Domäne erfordern wie beispielsweise Finanzbuchhaltung oder Lagerbestandsverwaltung. Das Konzept eines Datenstroms hingegen adressiert Anwendungen, in denen Ereignisabläufe oder die Beziehung zwischen einzelnen Ereignissen betrachtet werden, z.B. um Aktienkurse zu analysieren oder bösartiges Benutzerverhalten zu identifizieren. Das Zugriffsparadigma – Pull- oder Push-basiert – ist dabei an das Datenmodell gebunden: Traditionelle Datenbanksysteme sind Pull-basiert und bieten keine Unterstützung für kontinuierliche Anfragen mit automatischer Ergebnisaktualisierung, wohingegen nativ Push-basierte Systeme zur Datenstromverwaltung nur begrenzte Möglichkeiten für den Zugriff auf Archivdaten bieten. Die Trennung zwischen diesen beiden Systemklassen bedingt sowohl hohe Komplexität als auch hohe Wartungskosten bei Anwendungen, die gleichzeitig Persistenz und Echtzeitbenachrichtigungen bei Ergebnisänderungen erfordern.

Zum Schließen der Kluft zwischen beiden Paradigmen wurde eine neue Klasse von Informationssystemen gebildet, welche die Collection-basierte Semantik von DBMSs mit dem Push-basierten Zugriffsmodell von DSMSs kombiniert: **Echtzeitdatenbanken** („real-time databases“, RTDBs) halten die vom Klienten angefragten Daten kontinuierlich mit dem Datenbankzustand synchron und übermitteln alle Änderungen proaktiv „in Echtzeit“, d.h. unmittelbar nach jeder Änderung. Wie traditionelle Datenbankanfragen werden **Echtzeitanfragen** in RTDBs über persistenten Collections ausgewertet, welche konsistente Momentaufnahmen der modellierten Domäne repräsentieren. Gleichzeitig liefern Echtzeitanfragen allerdings nicht nur ein einziges Ergebnis zurück, sondern auch kontinuierliche Aktualisierungen, ähnlich wie Anfragen im Datenstrommanagement.

2 Echtzeitdatenbanken für kontinuierliche Ergebnisaktualisierung

In der Vergangenheit wurde der Begriff „Echtzeitdatenbanken“ („real-time databases“) für spezialisierte Pull-basierte DBMSs verwendet, die eine Ausgabe innerhalb strikter Zeitvorgaben produzieren [Pu93] [AH98]. Im Rahmen dieser Arbeit sind **Echtzeitdatenbanken**

jedoch Systeme, die einen Push-basierten Zugriff auf Collections in einer Datenbank ermöglichen. Populäre Vertreter sind beispielsweise Firebase, Meteor und RethinkDB. Entsprechend bezeichnen wir mit **Echtzeitanfragen** Push-basierte Anfragen, welche der gleichen Collection-basierten Anfragesemantik wie gewöhnliche Datenbankanfragen folgen, aber zusätzlich zum initialen Ergebnis auch einen kontinuierlichen Strom an Aktualisierungen liefern. Die in unserem Fokus stehenden **Echtzeitanwendungen** sind also reaktive oder interaktive Applikationen, die dem Benutzer neue Informationen so schnell wie möglich zur Verfügung stellen, nachdem sie in der Datenbank festgeschrieben wurden. Es handelt sich hierbei um Nutzungsszenarien mit weichen zeitlichen Beschränkungen („soft timing constraints“) und explizit nicht um sicherheitskritische oder andere Anwendungen, die strikte Obergrenzen für die Reaktionszeiten festlegen, wie z.B. Flugkontrollsysteme oder Kernkraftwerkskontrollen [St88].

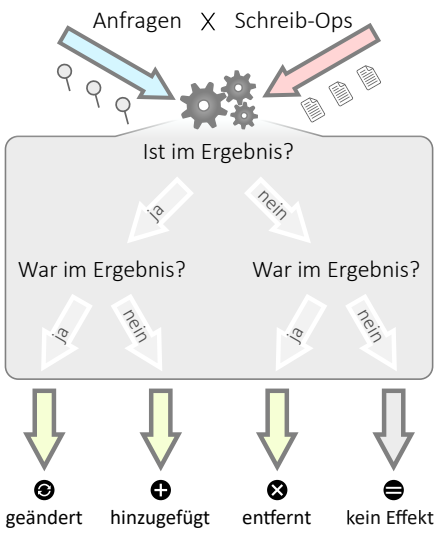


Abb. 1: Eine Echtzeitdatenbank muss jede Echtzeitanfrage (blauer Pfeil) mit jeder eingehenden Schreiboperation (roter Pfeil) abgleichen, um Ergebnisänderungen (grüne Pfeile) zu detektieren.

Konzeptuell haben die Informationen einer Echtzeitanfrage zwei Komponenten: das initiale Ergebnis und einen Strom an nachfolgenden Änderungsbenachrichtigungen. Das **initiale Ergebnis** entspricht den Daten, die von einer herkömmlichen ad hoc-Datenbankabfrage zurückgegeben würden, und erfasst somit die Datenelemente, die bei der Aktivierung der Anfrage mit dem Anfrageprädikat übereinstimmen. Anders als bei einer traditionellen Datenbankanfrage bleibt jedoch die Verbindung zwischen Klient und Datenbank bestehen und es werden **Änderungsbenachrichtigungen** („change events“) für alle Ergebnisänderungen übermittelt. Auf Basis dieser Informationen kann der Klient das Ergebnis auf dem neuesten Stand halten.

Ergebnisänderungen für eine aktive Echtzeitanfrage werden dabei durch einen kontinuierlichen Prozess ermittelt, der für alle eingehenden Schreiboperationen überprüft, ob und wie sie das Ergebnis der Anfrage beeinflussen. Zur Illustration erläutern wir diesen Prozess hier für unsortierte Filteranfragen über einer einzigen Collection und verweisen auf [Wi19, Kapitel 3] für eine Betrachtung komplexerer Echtzeitanfragen mit Ergebnissortierung, Aggregationen und Joins.

Abbildung 1 veranschaulicht, wie dieser Prozess auf zwei einfache Fragen heruntergebrochen werden kann, die für jeden geschriebenen Datensatz beantwortet werden müssen:

1. Wird das Anfrageprädikat nach dem Schreibvorgang erfüllt? („Ist im Ergebnis?“)
2. War das Anfrageprädikat vor dem Schreibvorgang erfüllt? („War im Ergebnis?“)

Wenn der Datensatz nach der Schreiboperation das Prädikat der Anfrage erfüllt (linker Zweig im Entscheidungsbaum), gibt es nur zwei Möglichkeiten: Entweder war der Daten-

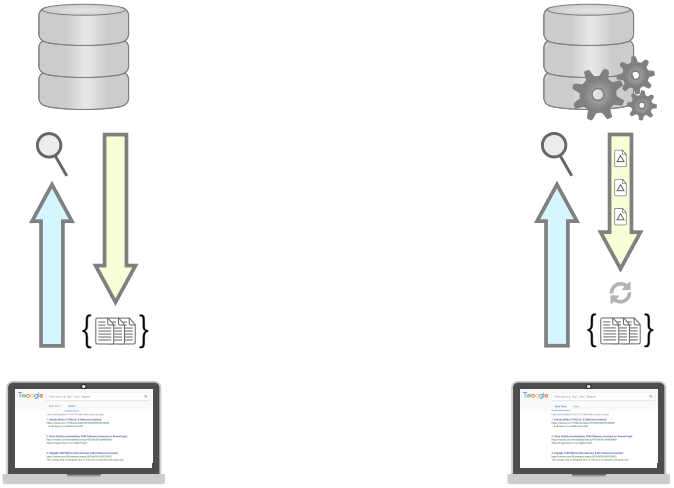
satz bereits Teil des Ergebnisses und wurde lediglich *geändert* oder er wurde durch die Schreiboperation zum Ergebnis *hinzugefügt*. Ähnlich verhält es sich, wenn ein Datensatz nach einer Schreiboperation nicht in das Anfrageergebnis fällt (rechter Zweig): Entweder war der Datensatz vor der Schreiboperation Teil des Ergebnisses und wurde folglich gerade *entfernt* oder die Schreiboperation hatte *keinen Effekt*.

Bei vielen aktiven Echtzeitanfragen oder hohem Schreibdurchsatz wird dieser kontinuierliche Prozess sehr aufwendig. Um den Ressourcenbedarf zu veranschaulichen, betrachten wir eine Applikation mit 1 000 aktiven Benutzern und einem durchschnittlichen Durchsatz von 1 000 Schreiboperationen pro Sekunde. Selbst bei nur einer Echtzeitanfrage pro Benutzer und bei sehr simplen Filteranfragen mit nur einem Prädikat muss die Echtzeitdatenbank bereits 1 Million Vergleichsoperationen durchführen – pro Sekunde. Für komplexe Anfragen ist der **Aufwand** noch größer, beispielsweise wenn ein bestimmter Sortierschlüssel zur Auswertung von Limit- oder Offset-Klauseln beachtet werden muss oder wenn Aggregationen oder Joins durchgeführt werden müssen. Durch gewisse „Optimierungen“ (z.B. Stapelverarbeitung) kann der Durchsatz auf Kosten der Latenz verbessert werden. Ebenso lässt sich durch effiziente Indizierung bei manchen Prädikaten eine bessere als die hier angenommene quadratische Komplexität erreichen. Wenn jedoch minimale Latenz zwingend erforderlich ist, gibt es für ein Echtzeitdatenbanksystem keine Alternative dazu, jede Schreiboperation im Kontext jeder aktiven Echtzeitanfrage zu betrachten. Eine skalierbare Implementierung der Anfrageauswertung ist daher essenziell, um Echtzeitdatenbanken in der Praxis nutzbar zu machen.

3 Anwendungsentwicklung mit Echtzeitanfragen

Es gibt zwei verschiedene Arten von Echtzeitanfragen, die sich darin unterscheiden, wie Daten der Anwendung zur Verfügung gestellt werden: **Ereignisstromanfragen** („event stream queries“) geben direkt das initiale Ergebnis und danach die unbearbeiteten Änderungsbenachrichtigungen an die Anwendung weiter, so dass diese das Ergebnis aktualisieren oder Geschäftslogik anwenden kann. **Selbsterneuernde Anfragen** („self-maintaining queries“) sind abstrakter, da sie die Ergebnisaktualisierung für die Anwendung transparent machen und bei jeder Änderung das vollständige (aktualisierte) Anfrageergebnis zur Verfügung stellen. Dadurch kann reaktives Verhalten in der Anwendung implementiert werden, ohne dass in der Geschäftslogik Konzepte zur Verarbeitung von Datenströmen oder Änderungsdeltas umgesetzt werden müssen. Durch Einsatz von selbsterneuernden Anfragen können einige statische Anwendungen somit in Echtzeitanwendungen umgewandelt werden, ohne signifikante Änderungen am Anwendungscode vorzunehmen.

Bei Anfrageschnittstellen, die auf Callback-Funktionen basieren, ist der Übergang zwischen statischem und Echtzeitverhalten besonders fließend. Dabei wird eine Anfrage asynchron ausgeführt und der Main-Thread der Anwendung wartet nicht auf das Eintreffen des Ergebnisses. Stattdessen wird mittels Callback-Funktion vorab festgelegt, wie das Ergebnis verarbeitet werden soll, wenn es schließlich eintrifft: Es wird dann nur noch die Callback-Funktion mit dem Ergebnis als Argument aufgerufen. Wenn die Anfrage als statische ad hoc-Anfrage ausgeführt wird (siehe Abbildung 2a), wird die Callback-



(a) Eine statische ad hoc-Anfrage liefert nur ein einziges Ergebnis, welches den Datenbankzustand zum Zeitpunkt der Anfrage repräsentiert.

(b) Eine selbsterneuernde Anfrage liefert eine Sequenz von Ergebnissen, von denen jedes einzelne die jeweils letzte Änderung widerspiegelt.

Abb. 2: Während eine statische ad hoc-Anfrage *Pull-basiert* ist und somit nur einmalig Informationen an den ausführenden Klienten zurückgibt, liefert eine selbsterneuernde Echtzeitanfrage nach jeder Änderung *proaktiv* eine aktualisierte Instanz des Ergebnisses aus.

Funktion nur einmal aufgerufen. Wenn die Anfrage als selbsterneuernde Echtzeitanfrage ausgeführt wird (siehe Abbildung 2b), wird die Callback-Funktion dagegen einmal aufgerufen, wenn das ursprüngliche Ergebnis zurückgegeben wird, und dann wiederum jedes Mal, wenn sich die zugrundeliegenden Daten geändert haben. Die Anwendung verhält sich also effektiv so, als ob eine neue statische Anfrage immer dann ausgeführt würde, wenn sie ein anderes Ergebnis als der letzte Aufruf zurückgeben würde. Als einzige Anforderung der Anwendung muss dazu jede Callback-Funktion idempotent implementiert werden, so dass ein Aufruf alle Auswirkungen des vorherigen Aufrufs überschreibt. Wenn die Callback-Funktion beispielsweise[3], das Ergebnis einer benutzerdefinierten Suche in ein HTML-Element rendert und dabei das zuletzt gerenderte Ergebnis ersetzt, verhält sich die Anwendung sowohl bei Pull-basierten ad hoc-Anfragen als auch bei Push-basierten selbsterneuernden Anfragen semantisch korrekt. Würde das Suchergebnis hingegen lediglich angehängt (ohne vorherige Entfernung des zuletzt gerenderten Ergebnisses), wäre die Darstellung ab dem zweiten Aufruf der Callback-Funktion nicht mehr sinnvoll.

4 Problemstellungen & Forschungsfrage

Konzeptuell können Echtzeitdatenbanken als Erweiterung zu traditionellen Datenbanksystemen verstanden werden, da sie der gleichen Anfragesemantik folgen, aber zusätzlich zu Pull- auch Push-basierten Zugriff bieten. In der Praxis hat sich jedoch keine Architektur für Echtzeitdatenbanken bewährt, die auf einem existierenden System aufsetzt. Da heutige

[3] Für eine Demonstration der im Rahmen dieser Arbeit entwickelten Implementierung siehe [WGR19b].

Vertreter meist von Grund auf neu entwickelt wurden, profitieren sie daher auch nicht vom reichhaltigen Funktionsumfang und der Stabilität, die einige Pull-basierte Systeme über Jahrzehnte erworben haben. Bisher wurde kein Mechanismus für die Bereitstellung von Push-basierten Echtzeitanfragen entwickelt, der jede der folgenden Challenges löst:

- **C_1 Skalierbarkeit.** Die Bereitstellung einer Echtzeitanfrage ist ressourcenintensiv, da sie die kontinuierliche Überwachung aller Schreibvorgänge erfordert, die das Anfrageergebnis beeinflussen könnten. Für erhöhte Skalierbarkeit verteilen heutige Echtzeitdatenbanken dabei die Anfragen oft über mehrere Server, sodass jeder Server nur für eine Teilmenge aller Anfragen verantwortlich ist. Keines der heutigen Systeme verteilt jedoch gleichzeitig auch die Schreiblast und daher muss jeder Server stets alle eingehenden Schreiboperationen verarbeiten: Der maximale Schreibdurchsatz des Gesamtsystems bleibt somit limitiert durch die Kapazität der einzelnen Server.
- **C_2 Ausdrucksmächtigkeit.** Die meisten APIs (Application Programming Interfaces) für Echtzeitanfragen sind im Vergleich zu ihren Pull-basierten Pendants stark limitiert. Aggregationen sind in der Regel nicht verfügbar und sortierte Anfragen werden oft nicht unterstützt oder haben starke Einschränkungen; so gibt es beispielsweise Implementierungen, die nur Sortierschlüssel mit einem einzigen Attribut erlauben oder eine Limit-, aber keine Offset-Klausel anbieten. Das Fehlen derartig grundlegender Funktionalität auf der Datenbankseite macht selbst bei relativ simplen Zugriffsmustern ineffiziente Workarounds im Anwendungscode erforderlich, weil Teile der Anfrageverarbeitung klientenseitig stattfinden müssen.
- **C_3 Rückwärtskompatibilität.** Heutige Echtzeitdatenbanken basieren auf Neuentwicklungen oder NoSQL-Systemen, die keinen Standards bzgl. Datenmodell oder Anfragesprache folgen. Sie implementieren proprietäre Protokolle für Pull- und Push-basierten Datenzugriff gleichermaßen und die Schnittstellen unterschiedlicher Systeme sind untereinander weitgehend inkompatibel. Ebenso lassen sie sich nur schwer mit traditionellen Datenbanken integrieren, sodass bestehende Anwendungen nur mit sehr hohem Aufwand um Push-basierte Anfragen erweitert werden können.
- **C_4 Abstrakte Schnittstellen.** Viele APIs für Echtzeitanfragen exponieren Eigenheiten der zugrundeliegenden Implementierung und bieten so nur eine geringe Datenunabhängigkeit. Viele Echtzeitdatenbanken bieten etwa keine automatische Ergebnisaktualisierung (vgl. selbsterneuernde Anfragen) und erfordern daher ein Verständnis von Systeminterna wie der Struktur und Semantik von Änderungsbenachrichtigungen. Derartige Schnittstellen sind hochgradig ineffizient, da sie Entwickler zum Lösen von Problemen außerhalb der eigentlichen Anwendungsdomäne zwingen.

In dieser Arbeit vertreten wir die Auffassung, dass keine der obigen Einschränkungen inhärent mit der Herausforderung verbunden ist, Push-basierte Echtzeitanfragen über Datenbank-Collections bereitzustellen. Wir gehen daher folgender Forschungsfrage nach:

Forschungsfrage: *Wie können ausdrucksmächtige Push-basierte Echtzeitanfragen für eine existierende Pull-basierte Datenbank skalierbar und generisch bereitgestellt werden?*

Um unseren Standpunkt zu belegen und die obige Forschungsfrage zu beantworten, entwickeln und implementieren wir ein generisches Systemdesign für Echtzeitdatenbanken,

welches Herausforderungen C_1 bis C_3 löst. Anschließend adressieren wir Herausforderung C_4, indem wir den Prototypen in eine bestehende Datenbank-Middleware integrieren und ihre rein Pull-basierte Anfrageschnittstelle um Push-basierte Echtzeitanfragen erweitern.

5 Hauptbeiträge der Arbeit

Wir präsentieren vier Hauptbeiträge. Unser erster Beitrag ist eine **Klassifikation** der Push-basierten Datenzugriffsmechanismen im heutigen Datenmanagement. Durch die Identifikation und Analyse der Mängel am aktuellen Stand der Technik leiten wir wichtige Erkenntnisse für unser Design ab und motivieren unsere Arbeit weiter. Unser zweiter Beitrag ist das **Systemdesign** InvaliDB, welches Push-basierte Echtzeitanfragen auf einer rein Pull-basierten Datenbank ermöglicht. InvaliDB ist hochskalierbar (vgl. C_1), unterstützt eine breite Palette an Anfragetypen (vgl. C_2) und ist als Erweiterung von traditionellen – rein Pull-basierten – Datenbanksystemen konzipiert (vgl. C_3). Unser dritter Beitrag besteht in der **Implementation** und ausführlichen experimentellen Evaluation unseres Designs. Als vierten Beitrag demonstrieren wir die Praktikabilität unseres Ansatzes durch die **Integration** des Prototyps mit der Pull-basierten Datenbank-Middleware Orestes. Das integrierte System bietet Entwicklern eine Schnittstelle für Zugriff auf die unbearbeiteten Änderungsbenachrichtigungen und eine für intuitiv nutzbare selbsterneuernde Anfragen (vgl. C_4).

- **H_1 Kategorisierung der heutigen Systemlandschaft.** Um ein intuitives Verständnis dessen zu vermitteln, was Echtzeitdatenbanken einzigartig macht, stellen wir ein Klassifikationsschema für Datenmanagementsysteme vor, das sich um die Unterstützung von Pull- und Push-basierten Anfragen in den jeweiligen Systemklassen dreht [WRG19]. Zu diesem Zweck vergleichen wir die verschiedenen Push-basierten Anfragemechanismen in traditionellen Datenbanksystemen, Echtzeitdatenbanken und Systemen zur Verwaltung und Verarbeitung von Datenströmen. Dabei heben wir jeweils die konzeptionellen Unterschiede zwischen Echtzeitdatenbanken und den anderen Systemklassen hervor. Darüber hinaus untersuchen wir die Limitationen bestehender Echtzeitdatenbanken und diskutieren, durch welche Designentscheidungen sie verursacht und wie sie vermieden werden können.
- **H_2 Systemdesign für skalierbare Echtzeitanfragen.** Aufbauend auf unseren Erkenntnissen präsentieren wir *InvaliDB* als ein skalierbares Systemdesign, welches Push-basierte Echtzeitanfragen auf einer Pull-basierten Datenbank ermöglicht. InvaliDB unterscheidet sich von bestehenden Systementwürfen durch (1) ein einzigartiges Verfahren zur Lastverteilung für *lineare Skalierbarkeit*, (2) Unterstützung für *ausdrucksmächtige Echtzeitanfragen* mit Ergebnissortierung, Aggregationen und Joins, (3) eine modulare Anfrage-Engine für *Datenbankunabhängigkeit* und (4) die *strikte Trennung* zwischen der Pull-basierten Datenbank und dem Subsystem für Echtzeitanfragen, um ihre Fehlerdomänen effektiv zu entkoppeln und eine unabhängige Skalierung zu ermöglichen.
- **H_3 Implementation auf MongoDB.** Um InvaliDBs Realisierbarkeit zu demonstrieren, entwickeln wir einen InvaliDB-Prototypen auf Basis des verteilten Stream Processors Storm und der Hauptspeicherdatenbank Redis für Push-basierte Echtzeitanfra-

gen auf der Pull-basierten Dokumentendatenbank MongoDB. In einer ausführlichen experimentellen Evaluation zeigen wir, dass der für eine InvaliDB-Instanz tragbare Durchsatz bei der Anfrageverarbeitung („sustainable query matching throughput“) linear mit der Anzahl der für die Anfrageverarbeitung eingesetzten Server skaliert, wobei die Ende-zu-Ende-Latenz über alle InvaliDB-Konfigurationen hinweg konstant niedrig bleibt und selbst bei Spitzenlast selten 100ms übersteigt.

H_4 Integration mit einer kommerziellen Datenbank-Middleware. Um die praktische Anwendbarkeit unseres Ansatzes zu demonstrieren, integrieren wir unseren Prototypen in die Datenbank-Middleware Orestes [Ge19] in der Quaestor-Architektur [Ge17]. Hier erfüllt InvaliDB zwei sehr unterschiedliche Aufgaben. Erstens stellt es für Klienten Push-basierte Echtzeitanfragen bereit, die von der Middleware selbst nicht unterstützt werden: Selbsterneuernde Anfragen sind leicht zu handhaben und eignen sich für Anwendungen, die lediglich eine aktuelle Sicht auf ihre kritischen Daten benötigen, während die hochflexiblen Ereignisstromanfragen für komplexere Zugriffsmuster wie benutzerdefinierte Echtzeit-Joins oder -Aggregationen geeignet sind. Zweitens ermöglicht InvaliDB ein konsistentes Caching der Ergebnisse von Pull-basierten Anfragen, indem es die Anfrageergebnisse kontinuierlich überwacht und die entsprechenden Caches bei einer Änderung sofort invalidiert. In einer weiteren experimentellen Evaluation zeigen wir, dass das durch InvaliDB ermöglichte Caching sowohl Latenz als auch Durchsatz der bestehenden Pull-basierten Anfrageschnittstellen um mehr als eine Größenordnung verbessert.

Unsere Klassifikation des heutigen Datenmanagement (vgl. H_1) bildet die Grundlage für ein Buch [WRG19] und verschiedene Tutorials [GWR17] [Wi18] [WGR19a]. Das Systemdesign InvaliDB (vgl. H_2), sein Prototyp (vgl. H_3) sowie die Quaestor-Architektur (vgl. H_4) wurden auf renommierten Konferenzen präsentiert und veröffentlicht, darunter die VLDB 2017 [Ge17], die BTW 2019 [WGR19b] und die VLDB 2020 [WGR20]. Das integrierte System wird von der Firma Baqend zur Entwicklung dynamischer Webseiten genutzt und seit Juli 2017 als Teil einer Backend-as-a-Service-Plattform kommerziell vertrieben. In diesem Kontext erfolgte weitere Forschung zur Beschleunigung von Webseiten, die u.a. in Buchform und auf der ICDE 2020 veröffentlicht wurde [GWR20] [Wi20].

6 Offene Fragen & Geplante Weiterentwicklung

Wir sind der festen Überzeugung, dass unsere Arbeit wertvolle Erkenntnisse für die Architektur von Echtzeitdatenbanken liefert, aber wir sehen noch viele Möglichkeiten für weitere Forschung und Entwicklung. Abschließend möchten wir einen Ausblick auf offene Herausforderungen und mögliche zukünftige Arbeiten in drei Kernbereichen geben.

Aggregationen & Joins. Unser Prototyp unterstützt bisher lediglich sortierte Filteranfragen über einzelne Collections, da dies der Anfragemächtigkeit vieler dokumentenorientierter NoSQL-Datenbanken entspricht (u.a. MongoDB, welches dem Prototypen zugrundeliegt). Zukünftige Arbeiten könnten unsere Implementierung um Anfragen mit Aggregationen und Joins erweitern oder Unterstützung für weitere Datenbanksprachen hinzufügen. Ebenso könnte eine Konsolidierung von InvaliDBs Konsistenzmodell („eventual

consistency“) mit strikteren Garantien verfolgt werden, z.B. durch die Realisierung transaktionaler Sichtbarkeit für Schreiboperationen.

Performance für Endbenutzer. In unserer Arbeit beschäftigen wir uns hauptsächlich mit der Infrastruktur im Backend, doch man sollte den Ressourcenverbrauch von Echtzeitanfragen für Endnutzer nicht aus den Augen verlieren. So könnte in nachfolgenden Arbeiten beleuchtet werden, welche Ressourcen eine Echtzeitanfrage beispielsweise für einen mobilen Nutzer tatsächlich erfordert und wie sie sich minimieren lassen. Eine Betrachtung aus der Verbraucherperspektive erscheint uns insbesondere deswegen relevant, weil Smartphones und andere Geräte mit eingeschränkter Rechenleistung, schwachen Netzwerkverbindungen und/oder begrenztem (monatlichem) Datenvolumen in bestimmten Nutzungsszenarien eher einen Flaschenhals darstellen können als Anwendungs- und Datenbankserver.

Anwendungsentwicklung. Wir sehen ein enormes Innovationspotenzial in der Aufwertung bestehender *Pull-basierter Schnittstellen* mit Push-basierten Funktionalitäten, da sich InvaliDB als zusätzliches Modul gut in bereits existierende Architekturen integrieren lässt. So könnten zum Beispiel bestimmte Teile einer Website interaktiv gemacht werden, um die Benutzererfahrung zu verbessern. Auch könnte InvaliDB eingesetzt werden, um ähnlich zu seiner Rolle innerhalb der Quaestor-Architektur veraltete Cache-Einträge mit geringer Latenz zu invalidieren und so die Datenaktualität zu verbessern.

7 Schlussgedanken

Zu Recht wurden Echtzeitdatenbanken in der Vergangenheit stets mit Vorsicht eingesetzt, da sie sich nur schwer in bestehende Anwendungen integrieren ließen und aufgrund mangelnder Skalierbarkeit nur für kleine Projekte genutzt werden konnten. Das in dieser Dissertation entwickelte Systemdesign InvaliDB adressiert diese Bedenken mit einer einzigartigen Kombination aus Merkmalen: Erstens abstrahiert es vom zugrundeliegenden Datenbanksystem und kann daher praktisch auf jeder Datenbank implementiert werden. Zweitens skaliert InvaliDB sowohl mit der Anzahl der nebenläufigen Anfragen als auch mit dem Schreibdurchsatz. Drittens – und wohl am wichtigsten für produktive Einsatzzwecke – ist InvaliDB als modulare Komponente mit einer isolierten Fehlerdomäne konzipiert und beeinträchtigt daher nicht die Ausfallsicherheit anderer Komponenten. Wir hoffen, dass unsere Arbeit neues Vertrauen in die Praxistauglichkeit von Echtzeitdatenbanken schafft und zu weiterer Forschung an diesem Thema innerhalb der Datenbank-Community anregt.

Literaturverzeichnis

[AH98] Andler, Sten F.; Hansson, Jörgen, Hrsg. Active, Real-Time, and Temporal Database Systems, Lecture Notes in Computer Science 1553. Springer Berlin Heidelberg, 1998.

[Ge17] Gessert, F.; Schaarschmidt, M.; Wingerath, W.; Witt, E.; Yoneki, E.; Ritter, N.: Quaestor: Query Web Caching for Database-as-a-Service Providers. VLDB, 2017.

[Ge19] Gessert, F.: Low Latency for Cloud Data Management. Diss., Univ. of Hamburg, 2019.

[GWR17] Gessert, F.; Wingerath, W.; Ritter, N.: Scalable Data Management: An In-Depth Tutorial on NoSQL Data Stores. In: BTW. 2017.

[GWR20] Gessert, F.; Wingerath, W.; Ritter, N.: Fast & Scalable Cloud Data Management. Springer Int. Publishing, 2020.

[GZ10] Golab, L.; Zsu, M. T.: Data Stream Management. Morgan & Claypool Publishers, 2010.

[He07] Hellerstein, J. M.; Stonebraker, M.; Hamilton, J. et al.: Architecture of a Database System. Foundations and Trends in Databases, 1(2), 2007.

[Pu93] Purimetla, B.; Sivasankaran, R.; Stankovic, J. et al.: A Study of Distributed Real-Time Active Database Applications. Bericht, University of Viginia, 1993.

[SC05] Stonebraker, M.; Cetintemel, U.: „One Size Fits All“: An Idea Whose Time Has Come and Gone. ICDE, 2005.

[SCZ05] Stonebraker, M.; Cetintemel, U.; Zdonik, S.: The 8 Requirements of Real-time Stream Processing. SIGMOD, 2005.

[St88] Stankovic, J. A.: Misconceptions About Real-Time Computing: A Serious Problem for Next-Generation Systems. Computer, 21(10), 1988.

[WGR19a] Wingerath, W.; Gessert, F.; Ritter, N.: NoSQL & Real-Time Data Management in Research & Practice. In: BTW. 2019.

[WGR19b] Wingerath, W.; Gessert, F.; Ritter, N.: Twoogle: Searching Twitter With MongoDB Queries. In: BTW. 2019.

[WGR20] Wingerath, W.; Gessert, F.; Ritter, N.: InvaliDB: Scalable Push-Based Real-Time Queries on Top of Pull-Based Databases (Extended). In: VLDB. 2020.

[Wi18] Wingerath, W.; Gessert, F.; Witt, E.; Friedrich, S.; Ritter, N.: Real-Time Data Management for Big Data. In: EDBT. 2018.

[Wi19] Wingerath, W.: Scalable Push-Based Real-Time Queries on Top of Pull-Based Databases. Dissertation, University of Hamburg, 2019.

[Wi20] Wingerath, W.; Gessert, F.; Witt, E.; Kuhlmann, H. et al.: Speed Kit: A Polyglot & GDPR-Compliant Approach For Caching Personalized Content. In: ICDE. 2020.

[WRG19] Wingerath, W.; Ritter, N.; Gessert, F.: Real-Time & Stream Data Management: Push-Based Data in Research & Practice. Springer Int. Publishing, 2019.

Wolfram „Wolle“ Wingerath wurde am 26. Januar 1988 in der wohl schönsten Ecke Deutschlands geboren (im Norden). Nach seinem Abitur 2007 hatte er Glück und bekam ein Vollstipendium der Studienstiftung des deutschen Volkes, sodass er sich bis zu seinem Master-Abschluss 2012 aufs Studium konzentrieren konnte. Während seiner anschließenden Promotion an der Uni Hamburg konzipierte er das skalierbare Design hinter der Echtzeitanfrage-Engine der Backend-as-a-Service-Plattform Baqend. Dabei entwickelte Wolle nicht nur einen soliden Hintergrund in Echtzeitdatenbanken, sondern auch den damit verbundenen Technologien wie Datenstromverarbeitung, NoSQL-Systemen, Cloud Computing und Big Data Analytics. Seit Januar 2018 arbeitet Wolle in Vollzeit bei Baqend und verantwortet dort heute als Data Engineer alle Belange rund um Datenanalyse und Echtzeitdatenverarbeitung. 2020 wurde Wolle für seine Arbeit als Junior-Fellow der Gesellschaft für Informatik ausgezeichnet. Weil Wolle eigentlich immer Lust auf neue Gesichter hat und weil er sich gerne mit anderen austauscht, referiert er regelmäßig auf Entwickler- und Forschungskonferenzen über spannende Erkenntnisse und Entwicklungen aus seinem Arbeitsumfeld.