Centrum voor Wiskunde en Informatica

REPORT*RAPPORT*

INS

Information Systems

INformation Systems

AmbientDB: relational query processing in a P2P network

P.A. Boncz, C. Treijtel

REPORT INS-R0306 JUNE 30, 2003

CWI is the National Research Institute for Mathematics and Computer Science. It is sponsored by the Netherlands Organization for Scientific Research (NWO).
CWI is a founding member of ERCIM, the European Research Consortium for Informatics and Mathematics.

CWI's research has a theme-oriented structure and is grouped into four clusters. Listed below are the names of the clusters and in parentheses their acronyms.

Probability, Networks and Algorithms (PNA)

Software Engineering (SEN)

Modelling, Analysis and Simulation (MAS)

Information Systems (INS)

P.O. Box 94079, 1090 GB Amsterdam (NL)
Kruislaan 413, 1098 SJ Amsterdam (NL)
Telephone +31 20 592 9333
Telefax +31 20 592 4199

ISSN 1386-3711

AmbientDB: Relational Query Processing in a P2P Network

Peter Boncz
P.Boncz@cwi.nl

Caspar Treijtel
C.Treijtel@cwi.nl

CWI
P.O. Box 94079, 1090 GB Amsterdam, The Netherlands

ABSTRACT

A new generation of applications running on a network of nodes, that share data on an ad-hoc basis, will benefit from data management services including powerful querying facilities. In this paper, we introduce the goals, assumptions and architecture of AmbientDB, a new peer-to-peer (P2P) DBMS prototype developed at CWI. Our focus is on the query processing facilities of AmbientDB, that are based on a tree-level translation of a global query algebra into multi-wave stream processing plans, distributed over an ad-hoc P2P network. We illustrate the usefulness of our system by outlining how it eases construction of a music player that generates intelligent playlists with collaborative filtering over distributed music logs. Finally, we show how the use of a Distributed Hash Tables (DHT) at the basis of AmbientDB to provide global indexing support allows applications like the P2P music player to scale to large amounts of nodes.

1998 ACM Computing Classification System: [E.1] Distributed Data Structures, [H.2.4] Data Models, Query Processing

Keywords and Phrases: Peer-to-Peer Systems, Relational Database Systems, Query Processing

Note: Work carried out under project INS1.2 "Database Architecture"

1. Introduction

Ambient Intelligence (AmI) refers to digital environments in which multimedia services are sensitive to people's needs, personalized to their requirements, anticipatory of their behavior and responsive to their presence. The AmI vision is being promoted by the MIT Oxygen initiative [12] and is becoming the focal point of much academic and commercial research. The AmbientDB project at CWI is performed in association with Philips, where the target is to address data management needs of ambient intelligent consumer electronics. In prototyping AmbientDB, CWI builds on its experience with DBMS kernel construction obtained in PRISMA [23] and Monet [4].

Figure 1 illustrates an example scenario, where a hypothetical ambient-intelligence enriched "amP2P" audio player automatically generates good playlists, fitting the tastes, probable interests and moods of the listeners present to the available music content. The amP2P player uses a local AmbientDB P2P DBMS to manage its music collection as well as the associated meta-information (among others how and when the music was played and appreciated).

We believe there is a common set of data management needs of AmI applications, and in the AmbientDB project we investigate new directions in DBMS architecture to address these. The ambitious end goal of AmbientDB is to become the pervasive data management infrastructure for these applications, analogous to what the RDBMS is to information systems. We highlight a number of possible uses of such data management infrastructure by AmI applications:

1. Uniting content collections stored in many sources.
2. Storing log information about media experiences and the perceived user reaction.
3. Analyzing logged information using machine learning techniques (e.g. data mining).

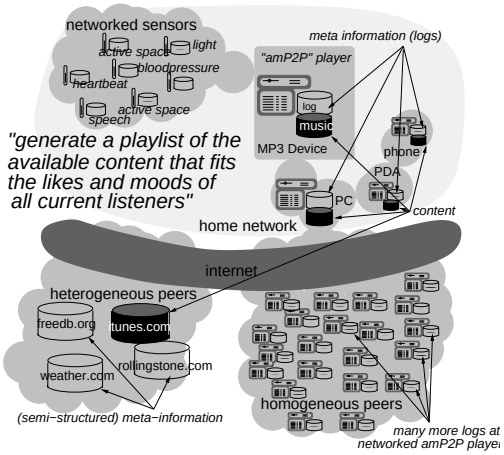

Figure 1: Music Playlist Scenario

```
create distributed table
AMP2P.USER (
    USERID varchar, PROFILE text)
primary key (USERID);

create distributed table
AMP2P.SONG (
    SONGID varchar, NAME varchar,
    ARTIST varchar, ALBUM varchar,
    LENGTH integer, FILENAME varchar)
primary key (SONGID);

create partitioned table
AMP2P.LOG (
    LOGID integer, SONGID varchar,
    USERID varchar, START daytime,
    DURATION integer)
primary key (LOGID);
```

Figure 2: Example Music Schema

4. Storing the results of machine learning (e.g. rule models) in a meta-repository.
5. Real-time evaluation of many such stored models to see whether some action should be performed.
6. Managing ontologies used for information retrieval.
7. Querying the direct environment for sensor input.

In the case of the amP2P player in Figure 1, the main idea is to exploit the wealth of knowledge about music preferences contained in the playlist logs of the thousands of online amP2P users, united in a global P2P network of AmbientDB instances. The amP2P player also uses AmbientDB to regularly query home sensors, such as active spaces that tell who is listening (in order to select music based on recorded preferences of these persons), and even use speech, gesticulation or mood detectors, to obtain feedback on the appreciation of the currently playing music. Additionally, it uses AmbientDB to query outside information sources on the internet to display additional artist information and links while music is playing, or even to incorporate tracks that can be acquired and streamed in from commercial music sites.

In this example, the shared music meta-information tables in all amP2P players is called *homogeneous*, as they all share a similar schema.[1] There may also be *heterogeneous* sources, that may either be AmbientDB instances with a different schema (such as the home sensors depicted in this scenario), or non-AmbientDB sources that export data through interfaces such as web services (SOAP) or ODBC. In case of non-AmbientDB sources such as the heterogeneous peers in Figure 1, the local AmbientDB on the MP3 player acts as a *wrapper* to these standard interfaces.

1.1 Goal

We define the goal for AmbientDB to provide full relational database functionality for standalone operation in autonomous devices, that may be mobile and disconnected for long periods of time. However, we want to enable such devices to cooperate in an ad-hoc way with (many) other AmbientDB devices when these are reachable. This justifies our choice for P2P, as opposed to designs that suppose a central server. We restate our motivation that by providing a coherent toolbox of data management functionalities (e.g. a declarative query language powered by an optimizer and indexing support) we hope to make it easier to create adaptive data-intensive distributed applications, such as sketched in the ambient-intelligent domain.

[1] The schemas need not be exactly identical, e.g. as multiple versions of the amP2P player will be floating around the internet at any point of time, each with a different version of the music database schema. There may even be other music players that export their logs in some different schema. As long as mappings for these various versions are available, AmbientDB will be able to hide these differences.

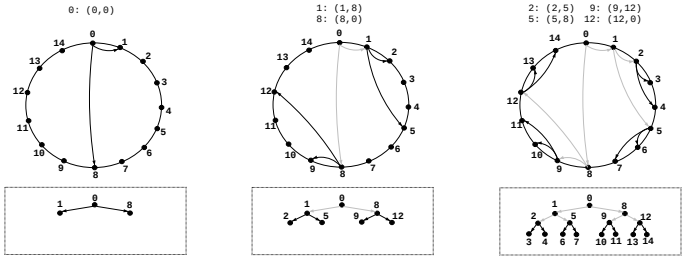

Figure 3: The construction of a query routing tree using Chord

In this paper, we propose a general architecture for AmbientDB, and focus on the issue of complex query processing in a potentially large and unreliable ad-hoc P2P network.

1.2 Assumptions

Let us summarize the main assumptions on which we base our work:

upscaling We assume the amount of cooperating devices to be potentially large (e.g. in agricultural sensor networks, or our example of on-line amP2P music players). Flexibility is our goal here, as we also want to use AmbientDB in home environments and even as a stand-alone data management solution. Also, we want to be able to cope with ad-hoc P2P connections, where AmbientDB nodes that never met before can still cooperate.

downscaling AmbientDB needs to take into account that (mobile) devices often have few resources in terms of CPU, memory, network and battery (devices can range from the PC down to the smart card [3] or even temperature sensors).

schema integration While we recognize that as different devices manage different semantics, this implies a need for heterogeneous schema re-mapping support (a.k.a. " model management" [6, 16]), in this paper we assume a situation where all devices operate under a common global schema. As the functionality of the AmbientDB schema integration component evolves, the schema operated on by the query processor will increasingly become a virtual schema that may differ widely from the local schema stored at each node.

data placement We assume the user to be in final control of data placement and replication. This assumption is made because in small mobile devices, users will want to have the final word on how scarce (storage) resources are used. This is a a main distinction with other work on distributed data structures such as DHTs [20, 22] and SDDS [14], where data placement is determined solely by the system.

network failure P2P networks can be quite dynamic in structure, and any serious internet-scale P2P system must be highly resilient to node failures. AmbientDB inherits the resilience characteristics of Chord [22] for keeping the network connected over long periods of time. When executing a query, it uses the Chord *finger* table to construct on-the-fly a *routing tree* for query execution purposes. Figure 3 shows a divide & conquer algorithm that creates new routing tree neighbors with the direct successor and a node halfway a given circular *finger-range*, divides this range among them and recursively continues in the neighbors. Such a routing tree only contains those nodes that participate in one user query and thus tends to be limited in size and lifetime. Our assumption is that while a query runs, the routing tree stays intact (a node failure thus leads to query failure, but the system stays intact and the query could be retried).

Since our work touches upon many sub-fields of database research[13, 9], we highlight the main differences. *Distributed database* technology works under the assumption that the collection of participating sites and communication topology is known a priori. This is not the case in AmbientDB. *Federated database* technology is the current approach to heterogeneous schema integration, but

```
RELEVANT :=
    select L.USERID, COUNT(*) as WEIGHT
    from   AMP2P.LOG L, AMP2P.SONG S
    where  L.SONGID = ''<normalized song name>'' and
           L.SONGID = S.SONGID and
           L.DURATION >= S.LENGTH
    group by USERID
VOTE :=
    select H.SONGID,  SUM(H.TIMESPLAYED * R.WEIGHT) AS VOTE
    from   AMP2P.HISTO H, RELEVANT R
    where  H.USERID = R.USERID
    groupby  H.SONGID
    order by VOTE descending
    limit 100
```

USERID	WEIGHT
uid5	92
uid9	72
..	..
uid2	2

SONGID	VOTE
sid44	4892
sid9	3472
..	..
sid87	342

Figure 4: Collaborative Filtering Query in SQL

is geared towards wrapping and integrating statically configured combinations of databases. In *mobile database* technology, one generally assumes that the mobile node is the (weaker) client, that at times synchronizes with some heavy centralized database server over a narrow channel. Again, this assumption does not hold here. Finally, P2P *file sharing* systems [17, 8] do support non-centralized and ad-hoc topologies. However, the provided functionality does not go beyond simple keyword text search (as opposed to structured DB queries).

1.3 Example: Collaborative Filtering in a P2P Database

In our example scenario, the intelligent "amP2P" player has access to a local content repository that consists of the digital music collection of the family (e.g. in mp3 and WMA formats). This collection – typically in the order of a few thousands of songs – is distributed among a handful of electronic devices owned by family members (PC, PDAs, mobile phones, mp3 players). These devices may have (mobile) access to the internet. The amP2P application would be based on an instance of AmbientDB running in each of these devices. The devices running AmbientDB form a self-organizing P2P network connecting the nodes for sharing all music content in the "home zone", and a possibly huge P2P network consisting of all amP2P devices reachable via the internet, among which only the meta-information is shared. To give an idea of the potential scale: the currently most popular music-oriented P2P file-sharing systems have 3 million nodes connected, which share 800 million songs (of which 1 million may be unique).

Figure 2 shows the schema created by the amP2P application consisting of three tables (`USER`, `SONG`, and `LOG`), that all appear in the global schema "`AMP2P`" in AmbientDB. Using these structures, the amP2P application registers which users are active on each device, and what music they play. These are *distributed* tables, which means that seen on the global level (over all devices connected in an AmbientDB network) it is formed by the union of all (overlapping) horizontal fragments of these tables stored on each device.

Our approach can be compared to a memory-based implicit voting scheme [5], where a prediction is made of the votes for the *active user* based on the given votes for other users. The vote $v_{i,j}$ corresponds to the vote of user i on item j. The predicted vote for the active user for item j is defined as $p_{a,j} = \overline{v}_a + \kappa \sum_{i=1}^{n} w(a,i)(v_{i,j} - \overline{v}_i)$, where $w(a,i)$ is a "weight" function defined on the active user and user i, $\overline{v}_i$ is the average vote for user i and κ is a normalizing factor.

We consider fully playing a song as a "vote" in this scenario, For approximating the weight function between the active user and another user, the active user chooses an *example song* as an input for the query. We define our weight function $weight(user_a, user_i)$ to be the times the example song has been *fully* played by user i. Figure 4 shows how this is expressed in AmbientDB: first we compute the listen count corresponding to the example song for all users in a temporary table, then we join this temporary table again with the log to compute the weighted vote for all songs, and take the highest 100 songs.

This formula could be refined in many ways, e.g. by also considering negative information (a skipped song, i.e. not fully played, is probably disliked), or giving songs that come right after sequence of skips extra weight. However, our focus here is not investigating multimedia retrieval techniques, but addressing the data management challenges they pose, for which this initial setup is sufficient.

1.4 Overview

In Section 2, we describe the general architecture of AmbientDB including its data model. In Section 2.2 we focus on query execution in AmbientDB, outlining its three-level query execution process, where global abstract queries are instantiated as global concrete queries (and rewritten) before being executed as (multi-) "wave" dataflow operator plans over a P2P routing tree. In Section 3 we describe how Distributed Hash Tables (DHTs) fit in AmbientDB as database indices on global tables, and show how these can optimize the queries needed by the amP2P music player. After outlining some of the many open challenges for further query optimization in AmbientDB and discussing related work, we conclude in Section 4.

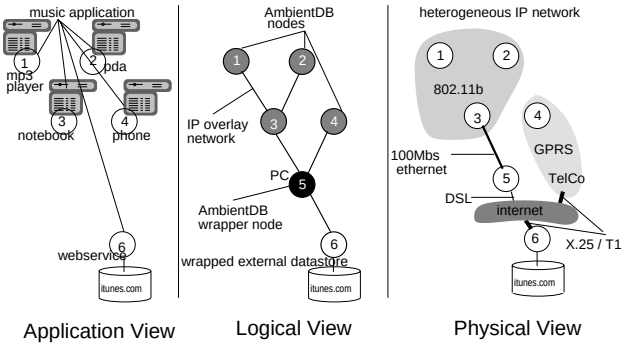

Figure 5: AmbientDB Routing Tree Using IP Overlay

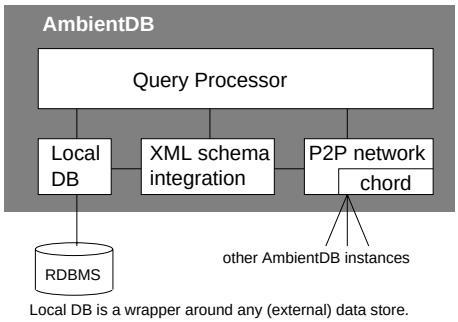

Figure 6: AmbientDB Architecture

2. AmbientDB Architecture

Figure 6 shows the AmbientDB architecture consisting of the following components:

1. *Distributed Query Processor:* gives applications the ability to execute queries on (a subset of) all ad-hoc connected devices as if it was a query to a central database on the union of all tables held on all selected devices. This is the focus of our paper and will be described more thoroughly in this section.

2. *P2P protocol:* Chord [22] is used to connect all AmbientDB nodes in a resilient way, and as the basis for implementing global table indices as Distributed Hash Tables (DHTs). Chord was selected as it provides a elegant, simple yet powerful DHT, with an open-source implementation as well as simulator available. At its core, Chord is a scalable lookup and routing scheme for possibly huge and P2P *IP overlay networks* made out of unreliable connections. On top of that, the AmbientDB P2P protocol adds functionality for creating temporary (logical) routing trees (see Figure 5). As shown in Figure 3, these routing trees, that are used for routing query streams, are subgraphs of the Chord network. When processing a query, we consider the *query node* the root. Such a tree may connect large amount of nodes that all participate in simultaneous bi-directional communication and query computation, while needing only a small number of connections per node (the tree fan-out).

3. *Local DB component:* Each AmbientDB node may store its own local tables, either internally in an embedded database, or in some external data source (e.g. RDBMS). In that case, the local DB component acts as a *wrapper* component, commonly used in distributed database systems [13]. The Local DB may or may not implement update and transaction support and its implemented query interface may be as simple as just a sequential scan. However, if more query algebra primitives are supported, this provides optimization opportunities during query processing.

4. *Schema integration engine:* different data delivered my different AmbientDB nodes need to be coupled using meta-data translations [6]. An often forgotten dimension here is providing support for schema evolution within one schema, such that e.g. old devices can be made

to work with newer ones. Note that AmbientDB itself does not attack the problem of constructing mappings between schemata automatically, but aims at providing the basic functionality for applying, stacking, sharing, evolving and propagating such mappings [16].

2.1 Data Model

AmbientDB provides a standard relational data model and a standard relational algebra as query language (thus it will be easily possible to e.g. create an SQL front-end). The queries that a user poses are formulated against *abstract* tables. The data in such tables may be stored in many nodes connected in the AmbientDB P2P network. A query may be answered only on the local node, or using data from a limited set of nodes or even against all reachable nodes. This explicit control over the set of participating nodes gives applications some way of providing a user with *converging answers*, e.g. by starting a query locally, and re-issuing it iteratively over more nodes, thereby refining the answer. Internally, an abstract table can take the following forms:

LT Each node has a private schema, in which *Local Tables* (LT) can be defined. Besides that, AmbientDB support global schemata that contain global tables T, of which all participating nodes N_i in the P2P network carry a table instance T_i. Each local instance T_i may also be accessed as a LT in the query node.

DT Over a set of nodes Q that participate in some global query, we define the *Distributed Table* (DT) as the union of local table instances at all nodes $T_Q = \texttt{union}(T_i)\ \forall i \in Q$. As we support ad-hoc cooperation of AmbientDB devices that never met before, tuples may be replicated at various nodes without the system knowing this beforehand.

PT The *Partitioned Table* (PT) is a specialization of the distributed table, where all *participating tuples* in each T_i are disjunct between all nodes. One can consider a PT a consistent snapshot view of the global table. A Partitioned Table has the advantage over a DT that exact query answers can often be computed in an efficient distributed fashion, by broadcasting a query and letting each node compute a local result without need for communication.

Whether a tuple participates in a partitioned table, is efficiently implemented by attaching a bitmap index (i.e. a boolean column) $T_i.Q$ to each local table T_i (see Figure 7). This requires little storage overhead, since adding one extra 64-bit integer column to each tuple suffices to support 64 partitioning schemes. Such partitioning schemes are typically only kept for the duration of a query, such that typically no more than a handful will coexist simultaneously anyway.

In AmbientDB, data placement is explicit, and we think that users sometimes need to be aware in which node tuples – stored in a DT/PT – are located. Thus, each tuple has a "virtual" column called `#NODEID` which resolves to the node-identifier (a special built-in AmbientDB type) where the tuple is located. This allows users to introduce location-specific query restrictions (return tuples only from the same node where some other tuples where found), or query tuples only from an explicit set of nodes.

2.2 Query Execution in AmbientDB

Query execution in AmbientDB is performed by a three level translation, that goes from the abstract to concrete and then the execution level (Figure 8). A user query is posed in the "abstract global algebra," which is shown in Table 1. This is an standard relational algebra, providing the operators for selection, join, aggregation and sort. These operators manipulate standard relational tables, and take parameters that may be (lists of) functional expressions. Lists are denoted (`List<Type>`), list instances <a,b,c>.

Any `Table` mentioned in the leaves of an abstract query graph, resolves to either a LT, DT, or PT (Figure 7). Thus, when we instantiate the parameters of an abstract relational operators, we get a *concrete* operator invocation. Table 2 shows the concrete operators supported in AmbientDB, where for reasons of presentation, each operator signature is simplified to consist only of the `Table` parameters and return type. The signatures are shown for all concrete instantiations of the

previously defined abstract operators plus two extra operators (partition and union), which are not present on the abstract level.

Starting at the leaves, an abstract query plan is made concrete by instantiating the abstract table types to concrete types. The concrete operators then obtained have concrete result types, and so the process continues to the root of the query graph, which is (usually) required to yield a local result table, hence a LT. Not all combinations of parameter instantiations of abstract operators are directly supported as concrete operators by AmbientDB, thus the AmbientDB query processor must use rewrite-rules to transform an abstract plan into a valid concrete query plan. AmbientDB does support the purely *local* concrete variant of all abstract operators, where all `Table`s instantiate to LTs. In combination with the concrete `Union`, which collects all tuples in a DT or PT in the query node into a LT, we see that any abstract plan can trivially be translated into a supported query plan: substitute each DT or PT by a $\texttt{union}_{merge}\texttt{(DT)}$, that creates an LT that contains all tuples in the DT.[2] This rewriting should be part of the rewriting done by a query optimizer that looks for an efficient plan.

Each concrete signature roughly corresponds with a particular query execution strategy. Apart from the purely local execution variants, the unary operators `select`, `aggr` and `order` support distributed execution (*dist*), where the operation is executed in all nodes on their local partition (LT) of a PT or DT, producing again a distributed result (PT or DT). On the execution level, this means that the distributed strategy broadcasts the query through the routing tree, after which each node in the tree executes the LT version of the operator on its fragment of the DT. Thus, in distributed execution, the result is again dispersed over all nodes as a PT or DT.

The *dist* aggregate hence produces distributed sub-results that only aggregate over a local partition of a DT or PT. The $\texttt{aggr}_{merge}$ variant is equivalent to $\texttt{aggr}_{local}\texttt{(union}_{merge}\texttt{(DT)):LT}$, but is provided separately, because aggregating sub-results in the nodes ("in network processing") reduces the fragments to be collected in the query node and can save considerable bandwidth [15].

Apart from the purely *local* join, there are three `join` variants:

- in the *broadcast* join, a LT at the query node is joined against a DT/PT, by broadcasting the LT, and joining it in each node with its local table.
- the *foreignkey* join exploits referential integrity to minimize communication. In AmbientDB, each node is in principle autonomous, which implies that viewed locally, its database should be consistent (have referential integrity). Therefore, join of a local table into a PT or DT over a foreign key, will be able to find all matching tuples locally. Thus, the operator can just broadcast the query and execute such joins locally (much like the *dist* strategy).

[2] The *elim* variant of union performs double elimination on some key. If a key is passed, both variants assume the local fragments to be ordered on this key, and produce an ordered result LT.

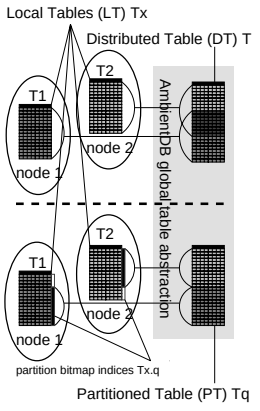

Figure 7: LT, DT and PT

abstract global algebra
`Select(Table t; Expr cond; List<Expr> result)→Table`
`Aggr(Table t; List<Expr> groupby, result)→Table`
`Join(Table left,right;Expr cond;List<Expr> result)→Table`
`Order(Table t; List<Expr> orderby, result)→Table`
`TopN(Table t; List<Expr> orderby, result; int limit)→Table`

data model
`Column(String name; int type)`
`Key(bool unique; List<Column> columns; Table table)`
`Table(String nme;List<Column> cols;List<Key> prim,forgn)`

expressions
`Expr(int type)`
`Expr::ConstExpr(String printedValue)`
`Expr::ColumnExpr(String columnName)`
`Expr::OperatorExpr(String opName, List<Expr>)`

Table 1: The Abstract Global Algebra

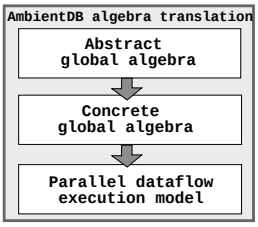

Figure 8: 3-level Algebra Translation

concrete global algebra (T_1, $T_2 \in$ {DT, PT})		
Union(T_1 t; List<Expr> key, result)→LT # *create LT from DT/PT by merging*		
Partition(DT t; List<Expr> key)→PT # *create PT from DT by finding duplicates*		
$select_{local}$(LT)→LT	$join_{local}$(LT,LT)→LT	$aggr_{local}$(LT)→LT
$select_{dist}$(T_1)→T_1	$join_{broadcast}$(LT,T_1)→T_1	$aggr_{merge}$(T_1)→LT
$select_{chord}$(DHT)→LT	$join_{split}$(LT,T_1)→T_1	$aggr_{dist}$(T_1)→DT
$order_{local}$(LT)→LT	$join_{foreignkey}$(T_1,DT)→T_1	$union_{merge}$(T_1)→LT
$order_{dist}$(T_1)→T_1	$topn_{local}$(LT)→LT	$union_{elim}$(T_1)→LT

Table 2: The Concrete Global Algebra

- the *split* join is a variant from the broadcast join (between an LT and DT), where the join predicate contains a restriction on `#NODEID` (thus specifying exactly on which node a tuple should be located). In this case, the LT relation is not broadcasted through the spanning tree, rather it is split when forwarded at each node in a local part (for those tuples where the `#NODEID` resolves to the local node) and in a part for each child in the routing tree. This reduces bandwidth consumed from $O(T * N)$ to $O(T * log(N))$ (where T is the amount of tuples in the DT and N is the number of nodes).

The `partition` is a special operator that performs double elimination: it creates a PT from a DT by creating a tuple *participation bitmap* at all nodes. Such an index records whether that tuple in a DT participates in the newly defined PT, at the cost of 1 bit per tuple. Recall that in AmbientDB, we support ad-hoc querying over nodes that meet for the first time and do not know what is replicated where. In order to be able to use the *dist* operators, we should convert a DT to a PT (note that in some cases users won't care about exact query answers and may choose to run a *dist* operator on a DT anyway).

2.3 Dataflow Execution

AmbientDB uses dataflow execution [23] as its query processing paradigm, where a *routing tree* that connects all nodes through TCP/IP connections (see Figure 5) is used to pass bi-directional tuple streams. Each node may receive tuples from its parent, process them with regards to local data, and propagate (broadcast) data to its children. Also, the other way around, it may be receiving data from its children, merging them (using some operators) with each other as well as with local data and pass the resulting tuples back to the parent. When an AmbientDB query runs, it may cause multiple simultaneous such *waves*, both upward and downward (and apart from that, each node may be involved in multiple such queries that might originate from different nodes in the network).

The third translation phase for query execution in AmbientDB consists of translating the concrete query plan into *wave-plans*, where each individual concrete operator maps onto one or more waves. Each wave, in turn consists of a graph of *dataflow algebra operators* (this dataflow algebra

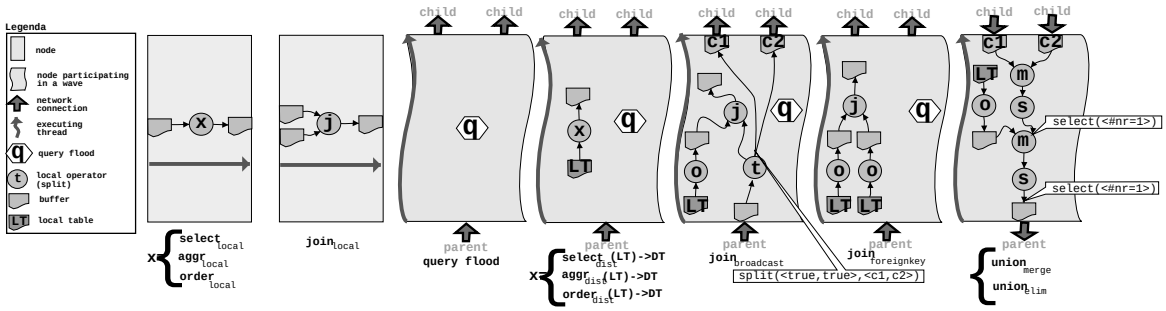

Figure 9: Wave Plans for most Concrete Operators

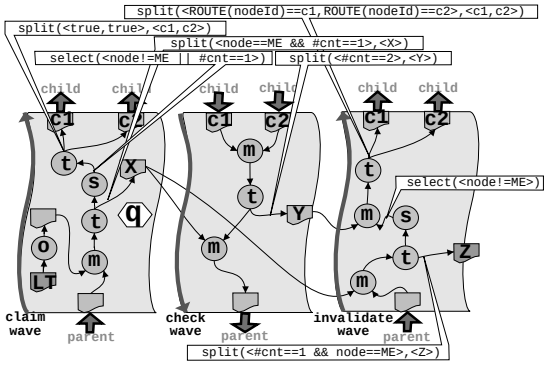

Figure 10: The 3-wave Partition Operator

	Dataflow Algebra Operators
n	**scan**(Buffer b)→Dataflow
s	**select**(Dataflow d; Expr cond; List<Expr> result)→Dataflow
a	**aggr**(Dataflow d; List<Expr> groupBy,aggr)→Dataflow
o	**order**(Dataflow d; List<Expr> orderby,result)→Buffer
n	**topn**(Dataflow d; List<Expr> orderby,result, int n)→Buffer
j	**join**(Dataflow d_l, d_r; List<Expr> key_l,key_r,result) # merge-join on dataflows ordered on key
m	**merge**(Dataflow d_l,d_r; List<Expr> key)→Dataflow # merges key-ordered dataflows, returning tuples in order # adds $t.\#cnt$: number of consecutive tuples with equal key # and $t.\#nr$: which ascends 0,1, etc.. in each such chunk
t	**split**(Dataflow d; List<Buffer>< $b_1..b_n$ >; List<Expr>< $f_1..f_n$ >)→Dataflow # returns equal stream, inserts $t \forall i : f_i(t) = true$ in b_i

Table 3: The Dataflow Algebra

is shown in Table 3). For brevity, in the plans we denote each dataflow operator by a single letter, and we uses an arrow going out of a buffer to denote an implicit `scan` operator. Dataflow operators may read multiple tuple streams but always produce just one output stream, such that each wave-plan can be executed by one separate thread, that calls to the root an object graph of nested *iterators*, where each iterator corresponds with a dataflow algebra operator (i.e. the Volcano iterator model [9]). The leaves of the dataflow query graphs either scan a LT, or read tuples from a neighbor node in the routing tree via a communication *buffer* (we consider a LT also a buffer). The total query wave plan consists of the union of all waves caused by concrete operators. The individual waves are connected by shared buffers holding LTs or DTs corresponding to the result of one concrete operator that is passed as parameter to the next.

In Figure 9, we show the mapping on wave plans for most concrete algebra operators. The left two rectangles depict the mapping for the purely *local* concrete operators, that read from local buffers and produce a new local buffer. The other wave plans in Figure 9 depict the *dist* variants of concrete operators. These distributed concrete operators typically broadcast their own query request first (depicted by a hexagon). Apart from that, the *dist* plans for `select`, `aggr`, `order`, `project` and the foreign-key `join` execute a buffer-to-buffer local operator in each node, without further communication. The broadcast `join`, however, produces a tuple stream through the network, sending a LT (which is assumed ordered on the join columns) from parent to all children, before executing the local join between this LT and the (ordered) local partition of a DT or PT.

The dataflow algebra operators shown in Table 3 use as algorithms resp. scan-select, quick-sort, merge-join, heap-based top-N and ordered aggregation, which are all stream-based and require little memory (at least no more than the memory used to hold the LTs present). These algorithms were chosen to allow the AmbientDB to run queries even on devices with little computational resources. In addition, AmbientDB uses producer/consumer *flow control* on all buffers, such that when a receiving buffer fills up, the producing thread blocks until it gets free again (in order to ensure that buffers occupy only a limited, preallocated amount of space).

Propagating a tuple stream to multiple children is done by the `split` operator, that receives a

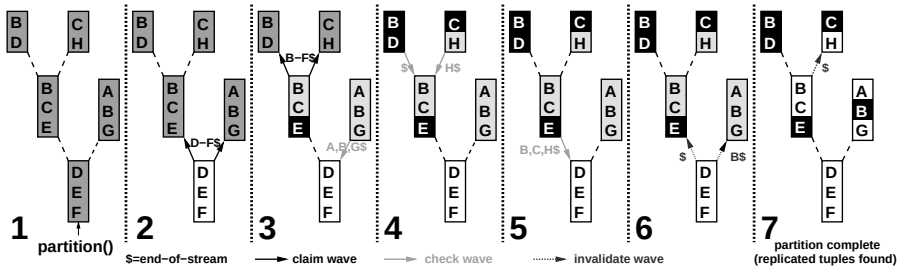

Figure 11: Illustration of Waves Generated By The Partition Operator (Only Key Values Shown)

list of boolean expressions and buffers, and puts the current tuple in each buffer for which the boolean resolves to true (thus for broadcasting to all children, a list of `true` constants is passed). The concrete $\texttt{union}_{merge}$ operator is implemented using the local `merge` operator, that merges two sorted streams, and adds artificial `#cnt` and `#nr` that tell how much equal elements of this value occur, and how many have passed already. The $\texttt{union}_{elim}$ additionally filters out duplicates using a plain `select` condition on this `#nr` column (it currently selects the first tuple).

Figure 11 shows that the `partition` operator (defined in Figure 10), maps on multiple waves. It consists of a *claim wave* that streams all unique key-values downward towards the leaves of the tree, tentatively assigning those local tuples to the local node for which the key was not yet received from the parent (i.e. turning on the bit corresponding to the that key-value in the – initially empty – local participation bitmap). As this is happening, in parallel a second *check wave* upward is started that consists of all such tentatively assigned key-values and node-ids (also in key-order), that is merged in each node in order to detect key values that were claimed twice in different branches of the tree. To correct those, a "winner" is determined (using e.g. the node-id that is nearest – or some other criterion), and in parallel a third *invalidate wave* downward is started, which against consists of of key,node-id tuples corresponding to the "losers", which are routed (much as in $\texttt{join}_{split}$) downward in the tree to the indicated node-id (on arrival, the bit in the participation bitmap of the key-value is switched off).[3]

In Table 4 a complex optimized join strategy is defined, which uses a semijoin-like reduction strategy [1, 2] to optimize the projection phase in a join. This way only those attribute values that are in the join result are fetched exactly once.

In the join, first the local the local partitions of a distributed table are ordered, with the *dist* strategy in temporary {`LPROJ,RPROJ`} buffers, that hold key value, join-columns, node-id and the other projection columns requested by the join. The union over only the table keys from all these buffers projection only key, join column and node-id, is then materialized in a local table at the query node, for both sides of the join. These are then joined together on join column to form a join-index (incrementally). While that is going on, the join-index is consumed by `project`s that filter out unique key values for both sides, and send these unique key values out to the children in the network, that respond with the projection columns. In this way, only those projection values for tuples that really participate in the join need to be sent over the network.

Actually, we see that we should have a $\texttt{join}_{split}$ to replace $\texttt{join}_{broadcast}$ here, as the projection requests are now broadcast to the entire network, while the join index contains the exact node-id where the tuple is located. Thus $\texttt{join}_{split}$ would not broadcast an LT, but partition it according to destination among its children.

The wave plans for this optimized join strategy are shown in Figure 12. Here, we show the advanced plan with targeted projection fetches as would have been produced by the $\texttt{join}_{split}$. As a final optimization remark, we note that the {`LPROJ,RPROJ`} intermediate buffers are now kept for the duration of the query in all nodes. An even more advanced optimization technique would start to partially free them, as the projection cursor advances in them. In Figure 12 we show an optimized join technique that

[3] In the wave-descriptions, we use the `#ME` constant to denote the private node-id of a node, and ROUTE(node-id) to compute the child node-id we should route into to reach that node.

$\texttt{join}_{semijoin}$(DT l,r; l.val==r.val; Expr $p_l|p_r$)→LT =
  $\texttt{join}_{local}$(RPROJ,$\texttt{join}_{local}$(LPROJ, JIDX, val=LPROJ.val and lkey==LPROJ.lkey, <rkey,val>|p_l),
                val ==RPROJ.val and rkey==RPROJ.rkey, $p_l|p_r$)

LORD = $\texttt{order}_{dist}$(l,<val,lkey>,<val,lkey,ME>|p_l)
RORD = $\texttt{order}_{dist}$(r,<val,rkey>,<val,rkey,ME>|p_r)
JIDX = $\texttt{join}_{local}$($\texttt{union}_{elim}$(LORD,<lkey>,<val,lkey,lnode>),
                $\texttt{union}_{elim}$(RORD,<rkey>,<val,rkey,rnode>),LORD.val==RORD.val,<lkey,rkey,val,lnode,rnode>)
LPROJ = $\texttt{union}_{merge}$($\texttt{join}_{broadcast}$($\texttt{project}_{local}$(JIDX,<lkey>,<lkey,val>),
                                LORD, val==LORD.val and lkey==LORD.lkey and lnode=ME,<lkey>|p_l),<lkey>,<lkey>|p_l)
RPROJ = $\texttt{union}_{merge}$($\texttt{join}_{broadcast}$($\texttt{project}_{local}$(JIDX,<rkey>,<rkey,val>),
                                RORD, val==RORD.val and rkey==RORD.rkey and rnode=ME,<rkey>|p_r),<rkey>,<rkey>|p_r)

Table 4: Optimized Semijoin reduction Join Project Strategy

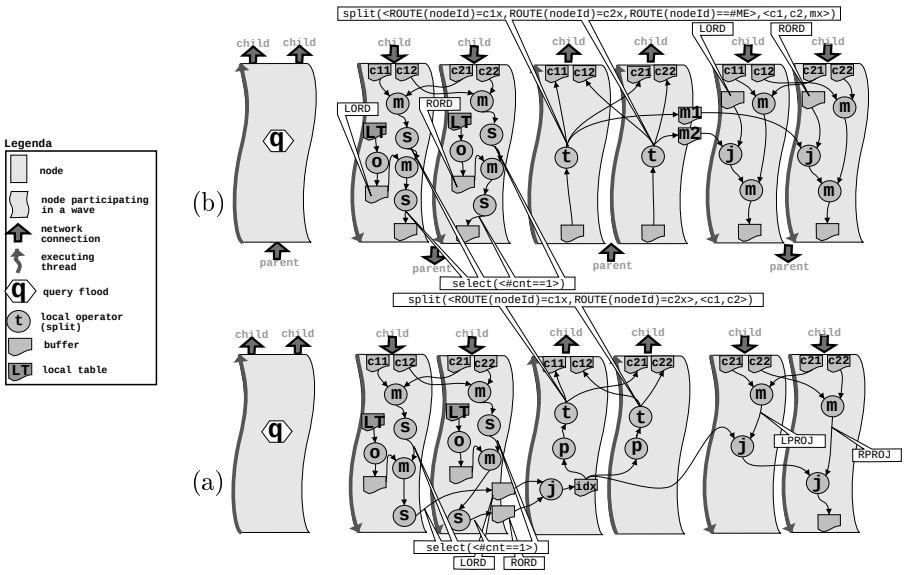

Figure 12: (a) Join waves for the query node (b) Join waves for all other nodes

2.4 Executing The Collaborative Filtering Query

Now we show an example of query execution in AmbientDB using the collaborative filtering query from Figure 4. Its translation into two concrete algebra queries is shown in Figure 13.

```
LT RELEVANT :=
   aggr_merge(DT T4 :=
      aggr_dist(DT T3 :=
         order_dist(DT T2 :=
            join_foreignkey(DT T1 :=
               select_dist(
                  DT L := AMP2P.LOG,
                  <L.USERID, L.SONGID, L.DURATION>,
                  <L.SONGID == ''<normalized song name>''),
               DT S := AMP2P.SONG,
               <T1.SONGID == S.SONGID AND
                T1.DURATION >= S.LENGTH>, <T1.USERID>),
            <T2.USERID>, <T2.USERID>),
         <T3.USERID>, <T3.USERID, TIMESPLAYED := count()>),
      <T4.USERID>, <T4.USERID, WEIGHT := sum(T4.TIMESPLAYED)>)

LT VOTE :=
   topn_local(LT T6 :=
      aggr_merge(DT T5 :=
         aggr_dist(DT T4 :=
            join_foreignkey(
               DT S := AMP2P.SONG,
               DT T3 := order_dist(DT T2 :=
                  join_broadcast(
                     LT R := RELEVANT R, DT T1 :=
                     order_dist(
                        DT L := AMP2P.LOG,
                        <L.USERID>, <L.USERID,L.SONGID>),
                     <R.USERID == T1.USERID>,
                     <T1.SONGID, T1.DURATION, R.WEIGHT>),
                  <T2.SONGID>,
                  <T2.SONGID, T2.DURATION, T2.WEIGHT>),
               <T3.SONGID == S.SONGID AND
                T3.DURATION >= S.LENGTH>,
            <T4.SONGID>, <T4.SONGID, VOTE := sum(T3.WEIGHT)>),
         <T5.SONGID>, <T5.SONGID, VOTE := sum(T4.VOTE)>),
      <T6.VOTE>, <T6.SONGID, T5.VOTE>, 100)
```

Figure 13: The example query in algebraic form

The RELEVANT query, that computes the relevance of each user, starts with a distributed select on the example song in the LOG DT. As also illustrated in Figure 15 the query is broadcast, and then in each node that stored the LOG DT, a selection is executed. This local result is streamed into a foreign-key join to the local SONG DT partition, in order to filter out those log records where the song was not played fully. Those result table fragments are then materialized in the $\texttt{order}_{dist}$ on USERID, which is required later for ordered aggregation. The aggregated values then start to stream back to the query node, passing through a $\texttt{aggr}_{merge}$ in each node that sums all partial results. As we may assume AMP2P users not to use many different devices, the duplicate elimination effect will be limited, such that the stream volume produced by a node will almost be the sum of all tuples received from the children.

The VOTE query then computes a relevance prediction for each song, by counting the times each user has (fully) listened to it and multiplying this by the just computer user's weight. We first broadcast-join RELEVANT with the LOG on all nodes, in order to attach the user's weight to each log-record. As

`RELEVANT` is ordered on `USERID`, we need to distributively sort `LOG` before this merge-join. The resulting DT fragments are then again distributively re-ordered on `SONGID` to allow quick foreign-key join into `SONG` (to exclude log entries for songs that were not played fully). Then, all weights are summed in an ordered aggregation that exploits the ordered-ness on `SONGID`, again in a distributed and merged aggregate. We take the top-100 of the resulting LT.

While this may seem already a complex query, we call this the "naive" strategy, as this solution will have scalability problems both with the number of users/nodes and number of songs. The first query will produce a large list of all users that have ever listened to the example song (not just those who particularly like the song), which will hog resources from all nodes in the network. The second query even takes more effort, as it will send basically all log records to the query node for aggregation! In the next section, we will describe how a slightly modified version of this query might be supported much more efficiently in an AmbientDB enriched with DHTs.

3. DHTs in AmbientDB

Distributed Hash Tables (DHTs) are useful lookup structures for large-scale P2P applications that want to query data spread out over many nodes, as a DHT allows to quickly reduce the amount of nodes involved in answering a query with high recall. This addresses an important scalability problem in P2P query processing, both because involving many nodes in answering a query decreases query performance (network communication tends to dominate cost) but also creates an overload in the average query frequency that each node needs to sustain (as the number of simultaneous queries increases linearly with the size of the network). In Gnutella-like P2P systems that do not use DHTs or other global indices, the total network therefore becomes limited by the slowest node (in network bandwidth, typically), and this is why such systems impose hard restrictions on the diameter of a search (using a TTL in query messages). Such a hard cut-off goes at the expense of recall, which in case of Gnutella means that it is easy to locate popular music, yet very difficult to locate less well-known songs [8].

In AmbientDB, an entire DT (or a subset of its columns) may be replicated in a *clustered index*. Our goal here is to enable the query optimizer to automatically accelerate selection queries using such DHTs. Getting the benefit of such advanced P2P structures via AmbientDB and its query language is a good example of the simplification of distributed application engineering we are after (programmers are currently forced to hardcode such accelerators in their application).

We use Chord [22] to implement such clustered indices as DHTs, where each AmbientDB node contains the index table partition that corresponds to it (i.e. the key-values of all tuples in the index partition hash to a *finger* that Chord maps on that node). As such, an AmbientDB DHT can be considered a special form of a PT, as the partitions are disjunct (with a system-determined placement policy). Invisible to users, DHT indices can be exploited by a query optimizer to accelerate lookup queries. We defined an additional concrete operator $\texttt{select}_{chord}\texttt{(DT):LT}$ that

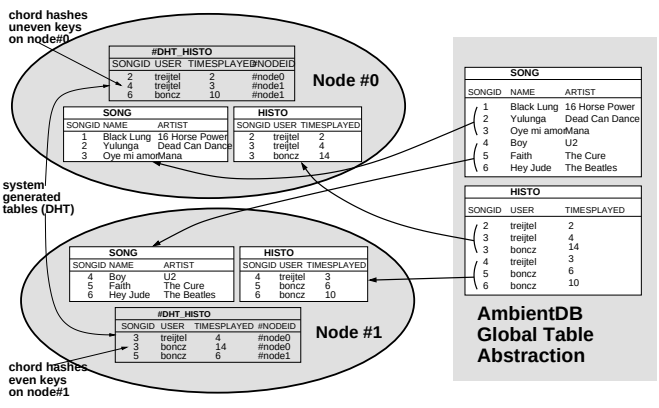

Figure 14: DT and DHT in AmbientDB

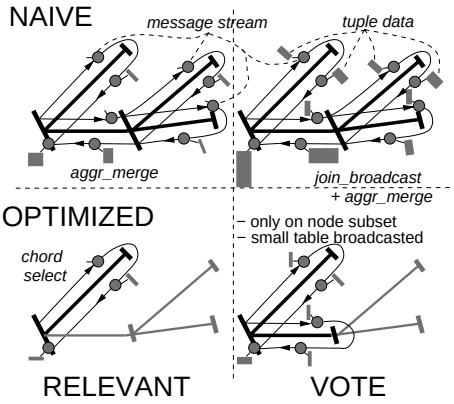

Figure 15: Network Bandwidth Compared

uses the DHT index on a DT to accelerate equi-selection on the index keys (see also Table 2). It is implemented in the dataflow level, by routing a message to the Chord finger on which the selection key-value hashes, and retrieving all corresponding tuples as an LT via a direct TCP/IP transfer.

As AmbientDB is intended to work even in fully ad-hoc encounters between nodes, the indices defined on distributed tables might be only partially filled at any point of time. Also, due to local resource constraints, the DHT may discard some of its inserts when it would exceed its maximum capacity at that node, so it may never be fully complete. Using a non-complete index for selecting in a table reduces the number of tuples found. At this moment, we decided that the AmbientDB end-user should decide explitly whether an index may be used, where a default behavior can be based on a *minimum coverage* threshold.[4].

3.1 Example: Approximated Collaborative Filtering

We now describe how the indexing feature of AmbientDB can be used to optimize the queries needed by our example amP2P music player to generate a playlist using collaborative filtering.

We introduce an extra `HISTO` as a static (off-line computed) histogram of fully-listened-to songs per user. Working with relatively stale log data should not be a problem for the playlist generation problem, as long as user the re-computation refresh rate is faster than the pace of change in human music taste. Thus, using `HISTO` instead of `LOG` directly reduces the histogram computation cost of our queries. Additionally, in the schema declaration of Figure 2, we created a *distributed index* on `HISTO`. As depicted in Figure 14, this leads to the creation of a DHT. The system-maintained indexed table is called `#DHT_HISTO` and carries an explicit `#NODEID` column for each tuple. As the indexed tuples were inserted by other nodes, this column is explict in a DHT rather than implict as in a LT, DT or PT.

```
create distributed table
AMP2P.HISTO(SONGID varchar, USERID varchar,
            TIMESPLAYED integer)
primary key (SONGID,USERID,#NODEID);

create distributed index
AMP2P.HISTO(SONGID,USERID,TIMESPLAYED) on (SONGID);
```

```
delete AMP2P.HISTO;
insert into AMP2P.HISTO
select L.SONGID, L.USERID, count(*) as  TIMESPLAYED
from AMP2P.LOG L, AMP2P.SONG S
where L.SONGID = SG.SONGID and
      L.DURATION >= SG.LENGTH
group by L.SONGID, USERID
order by L.USERID, SONGID;
```

Figure 16: Extensions To The Music Schema

Using this (indexed) `HISTO` table, we reformulate our query in Figure 17. First, we determine those 10 users that have listened most to the example song and retain their weight using a top-N query. The DHT makes this selection highly efficient: it involves just one remote node, and taking only the top-N listeners severly reduces the size of the result. We assume here that the influence extorted by listeners with a low weight can be discarded, and that a small sample of representative users is good enough for our playlist. Second, we ask for the explicit `#NODEID` locations of the selected relevant `HISTO` tuples. The reason for doing so is that we want to convey to AmbientDB that the last query involves a limited set of nodes such as to reduce the communication and query processing overhead in all other nodes. Note that this optimization also modifies the original query, as the user might have `LOG` entries on other nodes than those where he listened to the example song (though this is improbable). Third, we compute the weighted top-N of all songs on only those nodes as our final result.

Figure 18 shows the detailed concrete algebra plans for the optimized query plans. AmbientDB uses the $\texttt{select}_{chord}\texttt{(HISTO)}$ to directly identify the node with `HISTo` tuples of the `SONGID` of the example song, and retrieves `RELEVANT` (this is contrasted by a full network query and larger result in the naive strategy; as illustrated in Figure 15). It then locally computes the `RELEVANT_USERS` and `RELEVANT_NODES` in the query node as the top-10 listeners to that query resp. all nodes where those top listeners have listened to that song. Further queries will only concern this node subset, as we use the $\texttt{join}_{split}$ to route tuples only selectively to the `HISTO` DT, in order to compute the weighted scores for all (user,song) combinations in that subset, which are aggregated in two phases

[4] An AmbientDB node can use the percentage of its own local tuples having been inserted up-to-date in the index as an estimate of index coverage in the DT.

```
RELEVANT :=
    select USERID, SUM(TIMESPLAYED) as WEIGHT
    from   AMP2P.HISTO
    where  SONGID = ''<normalized song name>''
    group by USERID
    order by WEIGHT descending
    limit 10
```

USERID	WEIGHT
uid5	92
uid9	72
..	..
uid2	2

```
RELEVANTNODES :=
    select R.USERID, R.WEIGHT, H.#NODEID as LOCATION
    from   AMP2P.HISTO H, RELEVANT R
    where  H.SONGID = ''<normalized song name>'' and
           H.USERID = R.USERID
```

USERID	WEIGHT	LOCATION
uid5	92	node1
uid5	92	node2
uid9	72	node4
..	..	..
uid2	2	node1
uid2	2	node2

```
VOTE :=
    select H.SONGID, SUM(H.TIMESPLAYED * RN.WEIGHT) AS VOTE
    from   AMP2P.HISTO H, RELEVANTNODES RN
    where  H.USERID = RN.USERID
    groupby  H.SONGID
    order by VOTE descending
    limit 100
```

SONGID	VOTE
sid44	4892
sid9	3472
..	..
sid87	342

Figure 17: Optimized collaborative filtering query in SQL

(distributed and merge) to arrive at the predicted vote.

Though at this stage we lack experimental confirmation, it should be clear that performance is improved in the optimized strategy that only accesses 11 nodes w.r.t. the naive strategy that flooded the entire network twice with the entire contents `LOG` table (as also illustrated in Figure 15).

3.2 Future Work

At the time of this writing, AmbientDB is under full construction, with the first priorities being in the distributed query processor (including basic optimizer) and the networking protocol. The (Java) prototype being built is designed such that the codebase can be used both in the real DBMS as well as inside a network simulator (we are currently using NS2 [19]). In the near future, we hope to be able to obtain our first performance results on the query strategies described in this paper.

While we now use the $\texttt{join}_{split}$ to selectively route tuples from root to leaves, we intend to experiment with *on-the-fly* subset construction of routing trees. In our optimized example, all nodes of interest become available as a list of node-ids in `RELEVANT_NODES.LOCATION`. One can envision an trivial divide & conquer algorithm that selects some neighbours from this list (e.g. based on pinging a small random sample and taking the fastest ones), makes them your neighbours in the new IP overlay, divides the remaining node-list among them, and repeats the process there. Having a dedicated subnet may reduce the experienced latency by a factor log(N/M) (where N is tot total amount of nodes and M is the size of the subset), and even more importantly, reduces network bandwidth usage by a factor N/M. This creates a routing tree that is somewhat optimized to the latencies between peers, whereas a purely Chord-based network is randomly formed in that respect. Continuing on the

```
LT RELEVANT :=
   order_local(LT S :=
      select_chord(DT H :=
         AMP2P.HISTO,
         H.SONGID == ''<normalized song name>'',
         <H.USERID, H.TIMESPLAYED>),
      <S.USERID>,
      <S.USERID, S.TIMESPLAYED, LOCATION := H.#NODEID>)

LT RELEVANT_USERS :=
   topn_local(LT A :=
      aggr_local(LT R :=
         RELEVANT,
         <R.USERID>,
         <R.USERID, WEIGHT := sum(R.TIMESPLAYED)>),
      <A.WEIGHT>, <A.USERID, A.WEIGHT>, 10)

LT RELEVANT_NODES :=
   join_local(
      LT R := RELEVANT, LT U :=
      order_local(
         LT RELEVANT_USERS T,
         <T.USERID>, <T.USERID, WEIGHT>),
      <R.USERID == U.USERID>,
      <U.USERID, U.WEIGHT, R.LOCATION>)

LT VOTE :=
   topn_local(LT V :=
      aggr_merge(DT A :=
         aggr_dist(DT S :=
            order_dist(DT J :=
               join_split(
                  LT R := RELEVANT_NODES,
                  DT H := AMP2P.HISTO,
                  <H.USERID == R.USERID and
                     H.#NODEID == R.LOCATION>,
                  <H.SONGID,
                   SCORE := H.TIMESPLAYED*R.WEIGHT>),
               <J.SONGID>, <J.SONGID, J.SCORE>),
            <S.SONGID>, <S.SONGID, TOT := sum(S.SCORE)>),
         <A.SONGID>, <A.SONGID, VOTE := sum(A.TOT)>),
      <V.VOTE>, <V.SONGID, V.VOTE>, 100)
```

Figure 18: Optimized Query in Concrete Algebra

issue of creating a physical-network aware P2P logical networking structure, we may experiment with other DHT algorithms such as Pastry [21] and CAN [20] that take network proximity into account.

Another research issue is the exploitation of fixed connection patterns. In case of recurrent (dis-)connections, we might consider what happens if `partition` bitmaps would be cached and available for reuse in other queries (or transaction protocols). Also, we should investigate the cost of maintaining DHT indexing structures, and might take recurrence into account as a way to possibly improve maintenance performance. In any case, our current approach to DHTs is not resilient against failure (if a node leaves, his index partitions disappear), so some mechanism to replicate index partitions e.g. at multiple chord fingers, should be put in place.

The dataflow algorithms presented should be considered starting points and can be optimized in many ways. Since AmbientDB often sends streams of ordered key values over the network, it might also be worthwhile to investigate (lightweight) compression protocols to compress these streams in order to reduce the network bandwidth consumption (this is suggested in Figure 11 by sending some range-specifier instead of a full sequence of keys in the claim waves). A finally possible scalability improvement is to distinguish between powerful and weak nodes, putting only the powerful *backbone* nodes in the Chord ring. The weak *slave* nodes (e.g. mobile phone) would look for a suitable backbone and transfer all their data to it, effectively removing themselves as a bandwidth or CPU bottleneck from the routing tree.

3.3 Related work

Recently, database research has ventured into the area of *in-network query processing* with TinyDB [15]. The major challenge there is to conserve battery power while computing continuous (approximate) aggregate queries over an ad-hoc physical P2P radio network between a number of very simple sensors ("motes"). It turns out that executing the queries in the network (i.e. in the motes) is more efficient than sending individual messages to a base station from each mote, where many interesting optimization opportunities are opened by the interaction between the networking protocol and query processing algorithms. AmbientDB directly builds on this work, extending the ideas in TinyDB from aggregate queries only to full-fledged relational query algebra.

A second strain of recent related research are P2P data sharing protocols that go beyond Gnutella in scalability. Chord, Pastry and CAN [20, 21, 22] use distributed hash-tables (DHT) or other distributed hashing schemes to efficiently map data items into one node from a potentially very large universe of nodes. Thus, data stored in such networks is relocated upon entrance to the node where the hashing function demands it should be. While these algorithms are highly efficient, they are not directly applicable in AmbientDB, where data placement on a device is determined explicitly by the user. As described, AmbientDB can use DHTs for system-maintained index structures, that are not managed explicitly by the end-user.

Another effort of designing complex querying facilities in P2P systems is presented in [11]. In this paper an algorithm for join over a DHT is presented. This approach inserts all tuples to be joined on-the-fly in a CAN DHT [20], using a symmetric pipelined hash-join like [23]. The authors report performance problems, which may well stem from the avalanche of small messages that spread out over the system in such a join (in contrast to the stream-based approach in AmbientDB join).

In the database research community, there has been significant interest in Scalable Distributed Data Structures (SDDSs), such as LH* [14], with a focus on automatic scaling and load balancing in high query- *and* update-rate loads. An important difference with DHTs is that SDDSs make a distinction between clients and servers and are thus not "pure" P2P structures. Also, DHTs are designed on a bad network of unreliable connections, whereas SDDSs lack the resilience features that are necessary to stay connected in such harsh circumstances.

Some recent research has addressed the problem of heterogeneous schema integration in P2P systems. Bernstein et al. propose a formal framework called the *Local Relational Model* as a model for denoting manipulating schemas and mappings between them in P2P nodes [7]. In the case of the Piazza system, known techniques for (inverse) schema mappings in the relational domain are

extended to XML data models [10]. In future work on AmbientDB, we hope to build on this work for creating our XML schema integration component.

As for P2P database architecture work, we should mention PeerDB [18], which shares similar goals to AmbientDB, but with a focus on handling heteronegenous ad-hoc schemata. The system uses agent technology to implement extensible query execution. It matches schemas in an ad-hoc, Information Retrieval like approach, based on keywords attached to table and column names.

4. Conclusion

The major contribution of our research is a full query processing architecture for executing queries in a declarative, optimizable language, over an ad-hoc P2P network of many, possible small devices. We adopt dataflow execution on ordered streams and arrive at execution patterns that propagate in parallel multiple ordered tuple waves through the subset of all participating nodes. We have shown how Distributed Hash Tables (DHTs) can be incorporated seamlessly in the architecture to support efficient global indices that can transparently accelerate queries. In principle, data sharing and querying in the network is on a purely ad-hoc basis, where data can be replicated on the fine-grained tuple level. As such, we have advanced the state of the art in P2P systems.

That said, we see ample room for interesting future work. First off, we have identified here opportunities to further optimize the query processing strategies described, both in the area of query processing (join, partition, or distributed top-N), as well as in optimizing the networking protocol. In the next few months we will conduct first experiments and obtain performance results.

References

1. P.M.G. Apers, A.R. Hevner, and S.B. Yao. Optimization algorithms for distributed queries. *IEEE Transactions on Software Engineering*, 9(1):57–68, January 1983.
2. P.A. Bernstein and D.-M. W. Chiu. Using semi-joins to solve relational queries. *Journal of the ACM (JACM)*, 28(1):25–40, 1981.
3. C. Bobineau, L. Bouganim, P. Pucheral, and P. Valduriez. Picodbms: Scaling down database techniques for the smartcard. In *Proc. VLDB Conf.*, Cairo, Egypt, 2000.
4. P. Boncz. *Monet: A Next-Generation DBMS Kernel For Query-Intensive Applications*. PhD thesis, Universiteit van Amsterdam, Amsterdam, The Netherlands, May 2002.
5. J.S. Breese, D. Heckerman, and C. Kadie. Empirical Analysis of Predictive Algorithms for Collaborative Filtering. In *Proc. Conf. on Uncertainty in Artificial Intelligence*, July 1998.
6. A. Doan, P. Domingos, and A.Y. Halevy. Reconciling schemas of disparate data sources: a machine-learning approach. In *Proc. SIGMOD Conf.*, pages 509–520. ACM Press, 2001.
7. P.A. Bernstein et al. Data management for peer-to-peer computing: A vision. In *WebDB*, Madison, Wisconsin, June 2002.
8. Gnutella, http://www.gnutella.com/, April 2003.
9. Goetz Graefe. Encapsulation of parallelism in the volcano query processing system. In *Proc. SIGMOD Conf.*, pages 102–111, Atlantic City, New Jersey, United States, 1990. ACM Press.
10. S. Gribble, A. Halevy, Z. Ives, M. Rodig, and D. Suciu. What can peer-to-peer do for databases, and vice versa? In *WebDB 2001*, Santa Barbara, CA, USA, May 2001.
11. M. Harren, J.M. Hellerstein, R. Huebsch, B.T. Loo, S. Shenker, and I. Stoica. Complex queries in dht-based peer-to-peer networks. In *Proc. IPTPS Workshop*, Cambridge, MA, USA, March 2002.
12. S. Hedberg. Beyond desktop computing: Mit's oxygen project. *Distributed Systems Online*, 1, 2000.
13. D. Kossmann. The state of the art in distributed query processing. *ACM Computing Surveys (CSUR)*, 32(4):422–469, 2000.
14. W. Litwin, M.-A. Neimat, and D.A. Schneider. LH* – Linear Hashing for Distributed Files. In *Proc. SIGMOD Conf.*, May 1993.
15. S. Madden, M.J. Franklin, J.M. Hellerstein, and W. Hong. Tag: a tiny aggregation service for ad-hoc sensor networks. In *Proc. OSDI'02 Symposium*, December 2002.

16. S. Melnik, E. Rahm, and P.A. Bernstein. Rondo: A programming platform for generic model management. In *Proc. SIGMOD Conf.*, 2003.
17. Napster, http://opennap.sourceforge.net/, April 2003.
18. W.S. Ng, B.C. Ooi, K.-L. Tan, and A. Zhou. PeerDB: A P2P-based System for Distributed Data Sharing. In *Proc. ICDE Conf.*, March 2003.
19. The network simulator – ns-2, http://www.isi.edu/nsnam/ns/, April 2003.
20. S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Schenker. A scalable content-addressable network. In *Proc. SIGCOMM Conf.*, pages 161–172. ACM Press, 2001.
21. A. Rowstron and P. Druschel. Pastry: Scalable, distributed object location and routing for large-scale p2p systems. In *Proc. IFIP/ACM Middleware 2001*, November 2001.
22. I. Stoica, R. Morris, D. Karger, F. Kaashoek, and H. Balakrishnan. Chord: A scalable Peer-To-Peer lookup service for internet applications. In *Proc. SIGCOMM Conf.*, pages 149–160, 2001.
23. A.N. Wilschut and P.M.G. Apers. Dataflow query execution in a parallel main-memory environment. *Distributed and Parallel Databases*, 1(1):103–128, 1993.