AmbientDB: Relational Query Processing in a P2P Network

Peter Boncz and Caspar Treijtel

CWI, Kruislaan 413, 1098 SJ Amsterdam, The Netherlands
{P.Boncz,C.Treijtel}@cwi.nl

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 allows applications like the P2P music player to scale to large amounts of nodes.

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 [14] 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 [25] 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.

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

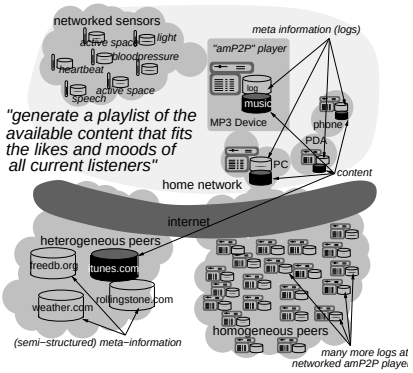


Fig. 1. Music Playlist Scenario

```
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);
```

Fig. 2. Example Music Schema

of the thousands of online amP2P users, united in a global P2P network of AmbientDB instances. The amP2P player could also use 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 could use 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.

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.

1.1 Assumptions

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 range from the PC down to smart cards [3] or temperature sensors).

schema integration: While we recognize that different devices manage different semantics which implies a need for heterogeneous schema re-mapping

support (a.k.a. " model management" [7, 18]), 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 [22, 8] and SDDS [16], 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 [8] 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. 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[15, 11], 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 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 [19, 23] 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).

```
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
```

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

```
 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
```

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

Fig. 3. Collaborative Filtering Query in SQL

1.2 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 [6]. 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, $\overline{v}_i$ 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 3 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.

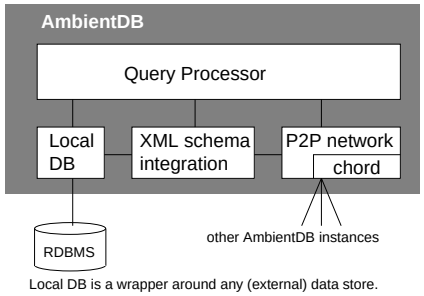
1.3 Overview

In Section 2, we describe the general architecture of AmbientDB. 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 and discussing related work, we conclude in Section 4.

2 AmbientDB Architecture

In the following, we describe the major architectural components of AmbientDB. The *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.

The *P2P Protocol* of AmbientDB uses Chord [8] to connect all 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 an 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 These routing trees, that are used for routing query streams, are subgraphs of the Chord network. When processing a query, we consider the node that issues the query as root.



The *Local DB* component of 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 [15]. 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 query optimization opportunities.

The *Schema Integration Engine* allows AmbientDB nodes to cooperate under a shared global schema even if they store data in different schemas, by using view-based schema mappings [7]. 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 automatically, but aims at providing the basic functionality for applying, stacking, sharing, evolving and propagating such mappings [18].

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 *global* 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.

Figure 4 shows that, a global table can be either an Local Table (LT), Distributed Table (DT) or Partitioned Table (PT). Each node has a private schema, in which *Local Tables* (LT) can be defined. Besides that, AmbientDB supports 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. The *Distributed Table* (DT) is defined over a set of nodes Q that participate in some global query, 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. 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 abstract 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 4). 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.

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 the concrete and then the execution level. 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 4). 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 in 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. 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 [17].

Apart from the purely *local* join, there are three `join` variants. The first is the *broadcast* join, a LT at the query node is joined against a DT/PT, by

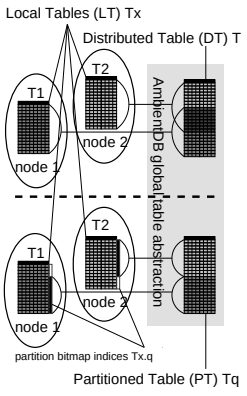


Fig. 4. LT, DT & PT

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

Concrete Global Algebra (T_1, $T_2 \in$ {DT, PT})		
Union(T_1 t; List<Expr> key, result)→LT # *merge DT/PT into a LT*		
Partition(DT t; List<Expr> key)→PT # *identifies 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_1,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

	Local Dataflow Algebra
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

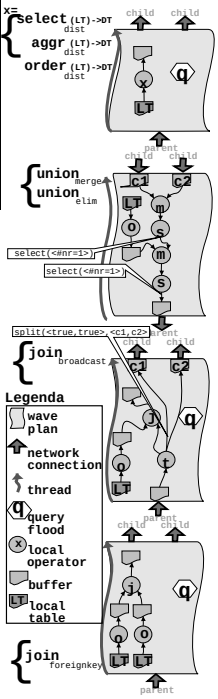


Fig. 5. Mappings

broadcasting the LT, and joining it in each node with its local table. Secondly, 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). The third variant is the *split* 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. Implementation details for partition can be found in [5].

2.3 Dataflow Execution

AmbientDB uses dataflow execution [25] as its query processing paradigm, where a *routing tree* that connects all nodes through TCP/IP connections 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.

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 *local dataflow algebra operators* (this dataflow algebra is shown in Table 3). For brevity, in the plans we denote each dataflow operator by a single letter, and we use 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 [10]). 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). Queries that consist of multiple waves pass data from one to the other using buffers that are shared between multiple waves (typically holding DT/PT fragments produced by a concrete operator that serves as input for the next).

In Figure 5, we show the mapping on wave plans for some concrete algebra operators. These operators typically broadcast their own query request first (depicted by a hexagon). The *dist* plans for `select`, `aggr`, `order` (top), and the foreign-key `join` (bottom) execute a buffer-to-buffer local operator in each node, without further communication. The broadcast `join` (middle), however, produces a tuple stream through the network. It splits the incoming stream containing the broadcasted LT to all its children, before executing the local join with its local partition, producing a result buffer in each node.

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.

2.4 Executing The Collaborative Filtering Query

Now we show an example of query execution in AmbientDB using the collaborative filtering query from Figure 3. Its translation into two concrete algebra queries is shown in Figure 6.

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 8 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.

```
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.DURATION >= S.LENGTH AND
           T1.SONGID == S.SONGID>, <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)
```

Fig. 6. Concrete Algebra Example Query

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 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 merge-joining it. The resulting DT fragments are again distributively reordered on SONGID to allow quick foreign-key join into SONG (to exclude songs that were not played fully). All weights are then 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 it 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.

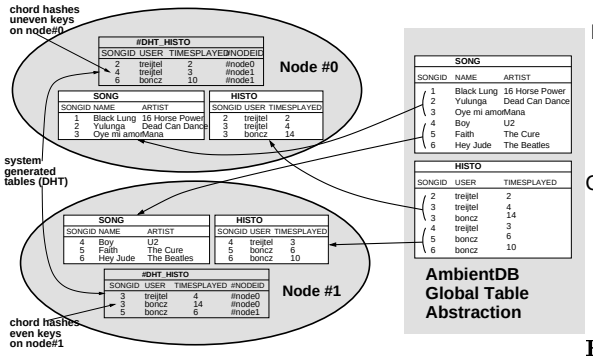


Fig. 7. DT and DHT in AmbientDB

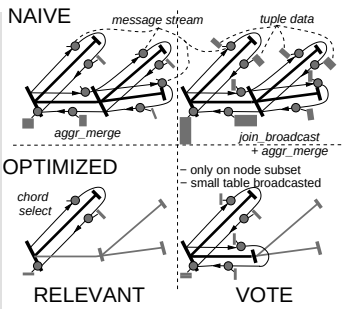


Fig. 8. Naive vs. DHT accelerated Network Bandwidth Usage

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. 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 queries using such DHTs. Getting the benefit of such advanced P2P structures via AmbientDB and its query language is an 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 [8] 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). 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 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 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.[1].

[1] 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

Fig. 9. Optimized collaborative filtering query in SQL

3.1 Example: Optimized 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 playlists.

```
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;
```

Fig. 10. Extensions To The Music Schema

We introduce an extra `HISTO` table as a pre-computed histogram of fully-listened-to songs per device. Working with somewhat 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 10, we created a *distributed index* on `HISTO`. As depicted in Figure 7, 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.

Using this (indexed) `HISTO` table, we reformulate our query in Figure 9. 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 11 shows the detailed concrete algebra plans for the optimized query plans. We use $\texttt{select}_{chord}$(HISTO) to directly identify the node with HISTo tuples of the SONGID of the example song, and retrieve RELEVANT (this is contrasted by a full network query and larger result in the naive strategy; as illustrated in Figure 8). Locally in the query node, we then compute the RELEVANT_USERS and RELEVANT_NODES tables as the top-10 listeners to that query resp. all nodes where those top listeners have listened to that song. Further queries 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 (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 8).

```
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 T :=
      RELEVANT_USERS,
      <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)
```

Fig. 11. Concrete Algebra Plan

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 [21]). In the near future, we hope to obtain our first performance results on the query strategies described here.

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 issue of creating a physical-network aware P2P logical networking structure, we may experiment with other DHT algorithms such as Pastry [24] that take network proximity into account.

The dataflow algorithms presented should be considered starting points and can be optimized in many ways. In [5] we show an optimized join technique that uses a semijoin-like reduction strategy [1, 2], to optimize the projection phase in a join such that only those attribute values actually in the join result are fetched exactly once. 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. 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, removing themselves as a bottleneck from the query routing tree.

3.3 Related work

Recently, database research has ventured into the area of *in-network query processing* with TinyDB [17]. 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, Oceanstore and CAN[22, 8] 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 [13]. 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 [22], using a symmetric pipelined hash-join like [25].

In the database research community, there has been significant interest in Scalable Distributed Data Structures (SDDSs), such as LH* [16], 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 [9]. In the case of the Piazza system, known techniques for (inverse) schema mappings in the relational domain are extended to XML data models [12]. 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 [20], 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, possibly 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. Apers, A. Hevner, and S. Yao. Optimization algorithms for distributed queries. *IEEE Transactions on Software Engineering*, 9(1):57–68, 1983.
2. P. Bernstein and D. 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.*, 2000.
4. P. Boncz. *Monet: A Next-Generation DBMS Kernel For Query-Intensive Applications.* PhD thesis, Universiteit van Amsterdam, May 2002.
5. P. Boncz and C. Treijtel. Ambientdb: Relational query processing in a p2p network. Technical Report INS-R0305, CWI, June 2003.
6. J. Breese, D. Heckerman, and C. Kadie. Empirical Analysis of Predictive Algorithms for Collaborative Filtering. In *Proc. Conf. on Uncertainty in Artificial Intelligence*, July 1998.
7. A. Doan et al. Reconciling schemas of disparate data sources: a machine-learning approach. In *Proc. SIGMOD Conf.*, pages 509–520, 2001.
8. I. Stoica et al. Chord: A scalable Peer-To-Peer lookup service for internet applications. In *Proc. SIGCOMM Conf.*, pages 149–160, 2001.
9. P. Bernstein et al. Data management for peer-to-peer computing: A vision. In *Proc. WebDB Workshop*, 2002.
10. G. Graefe. Encapsulation of parallelism in the volcano query processing system. In *Proc. SIGMOD Conf.*, pages 102–111, 1990.
11. G. Graefe. Volcano – an extensible and parallel query evaluation system. *IEEE TKDE*, 6(1):120–135, 1994.
12. S. Gribble, A. Halevy, Z. Ives, M. Rodig, and D. Suciu. What can peer-to-peer do for databases, and vice versa? In *Proc. WebDB Workshop*, 2001.
13. M. Harren, J. Hellerstein, R. Huebsch, B. Loo, S. Shenker, and I. Stoica. Complex queries in dht-based peer-to-peer networks. In *Proc. IPTPS Workshop*, 2002.
14. S. Hedberg. Beyond desktop computing: Mit's oxygen project. *Distributed Systems Online*, 1, 2000.
15. D. Kossmann. The state of the art in distributed query processing. *ACM Computing Surveys (CSUR)*, 32(4):422–469, 2000.
16. W. Litwin, M.-A. Neimat, and D. Schneider. LH* – Linear Hashing for Distributed Files. In *Proc. SIGMOD Conf.*, 1993.
17. S. Madden, M. Franklin, J. Hellerstein, and W. Hong. Tag: a tiny aggregation service for ad-hoc sensor networks. In *Proc. OSDI'02 Symposium*, 2002.
18. S. Melnik, E. Rahm, and P. Bernstein. Rondo: A programming platform for generic model management. In *Proc. SIGMOD Conf.*, 2003.
19. Napster, http://opennap.sourceforge.net/, 2003.
20. W. Ng, B. Ooi, K.-L. Tan, and A. Zhou. PeerDB: A P2P-based System for Distributed Data Sharing. In *Proc. ICDE Conf.*, 2003.
21. The network simulator – ns-2, http://www.isi.edu/nsnam/ns/, April 2003.
22. S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Schenker. A scalable content-addressable network. In *Proc. SIGCOMM Conf.*, pages 161–172, 2001.
23. Matei Ripeanu, Adriana Iamnitchi, and Ian Foster. Mapping the gnutella network. *IEEE Internet Computing*, 6(1):50–57, 2002.
24. A. Rowstron and P. Druschel. Pastry: Scalable, distributed object location and routing for large-scale p2p systems. In *Proc. IFIP/ACM Middleware 2001*, 2001.
25. A. Wilschut and P. Apers. Dataflow query execution in a parallel main-memory environment. *Distributed and Parallel Databases*, 1(1):103–128, 1993.