SQPR: Stream Query Planning with Reuse

Evangelia Kalyvianaki [#], Wolfram Wiesemann [#], Quang Hieu Vu [*], Daniel Kuhn [#], Peter Pietzuch [#]

[#]*Department of Computing, Imperial College London, United Kingdom*
{ekalyv, wwiesema, dkuhn, prp}@doc.ic.ac.uk

[*]*Institute for Infocomm Research, Singapore*
qhvu@i2r.a-star.edu.sg

***Abstract*—When users submit new queries to a distributed stream processing system (DSPS), a query planner must allocate physical resources, such as CPU cores, memory and network bandwidth, from a set of hosts to queries. Allocation decisions must provide the correct mix of resources required by queries, while achieving an efficient overall allocation to scale in the number of admitted queries. By exploiting overlap between queries and reusing partial results, a query planner can conserve resources but has to carry out more complex planning decisions.**

In this paper, we describe SQPR, a query planner that targets DSPSs in data centre environments with heterogeneous resources. SQPR models query admission, allocation and reuse as a single constrained optimisation problem and solves an approximate version to achieve scalability. It prevents individual resources from becoming bottlenecks by re-planning past allocation decisions and supports different allocation objectives. As our experimental evaluation in comparison with a state-of-the-art planner shows SQPR makes efficient resource allocation decisions, even with a high utilisation of resources, with acceptable overheads.

I. Introduction

An increasing number of applications require the processing of infinite streams of data in near real-time according to a large set of continuous queries. For this purpose, stream processing systems have been successfully used in financial data processing [1], network monitoring [2] and supply chain management applications [3]. *Distributed stream processing systems* (DSPSs) [1], [4], [5] partition the execution of queries across a set of hosts in order to achieve higher performance, in terms of supported data stream rates, and better scalability, in terms of the number of concurrent queries.

A challenge that all DSPSs face is *query planning*—the initial allocation of resources from hosts to queries. Individual queries require a mix of computational, memory and network resources from potentially multiple hosts in order to process data streams at a given target rate. Queries consist of query operators that consume CPU and memory resources on a host. In addition, operators also need network bandwidth to exchange streams with partial query results between hosts. Query planning should provide an allocation of resources that is efficient in order to support the maximum number of queries, while satisfying the performance goals of individual queries.

The query planning problem is challenging because of the large number of resources managed by the DSPS and the complex resource requirements of queries. A bad query planner may use resources inefficiently and therefore not admit new queries to the system when there is a shortage of some resource while others are under-utilised. Any of the resources may become a bottleneck in a data centre: for example, a host that has exhausted its network bandwidth to other hosts may be unable to support additional operators, irrespective of its spare CPU resources. Query planning should have the goal of load-balancing resource allocation across hosts to achieve uniform query performance [6]. However, in a data centre with virtualised hosts, it may also be desirable to skew the load distribution to switch off idle virtual machines.

Multi-query optimisation [7] has been shown to be a powerful technique for improving the efficiency of query processing. By exploiting the overlap between queries in terms of shared operators and streams, *query reuse* can save resources through reuse of partial results from sub-queries, thus increasing system scalability. However, query reuse complicates query planning, and simple heuristics may result in poor allocation decisions. For example, a strategy that always greedily reuses existing sub-queries may exhaust a host's available network bandwidth before its CPU resources are fully utilised. This introduces unnecessary hot-spots in the system and requires the query planner to revisit past planning decisions.

Most research in this space focuses on adaptive query optimisation of already running queries [4], [8], [9]. This assumes that initial query planning at deployment time is trivial due to abundant resources available to the DSPS. Such over-provisioning is wasteful especially in virtualised data centres where unused resources can be reclaimed for other applications. In addition, poor decisions on initial query planning are costly to correct at runtime using operator migration, which impacts query performance.

Existing proposals for initial query planning such as SODA [9] divide the problem into admission control and operator placement. This requires the flexibility to allocate fewer resources to queries after too many queries have been admitted to the system. This may not be possible for queries that have inelastic resource requirements, such as queries that process sensor data at a given source rate.

We describe *SQPR* (Stream Query Planning with Reuse), a new query planner for DSPSs that is designed to perform well in resource-scarce environments when initial query planning is non-trivial. SQPR efficiently allocates resources to queries, while exploiting overlap between queries for reuse. To achieve this, we formalise query planning as a single constrained optimisation problem to provide queries with fixed resources, while maximising resource utilisation and respecting allocation goals of the system, for example, in terms of

load-balancing. This holistic view combines query admission, operator placement and reuse into a single optimisation model.

A key feature of SQPR is that it revisits past planning decisions for existing queries as new queries are added to improve the efficiency of the resource allocation. To make query planning tractable with replanning, SQPR provides approximate solutions by only considering a subset of all queries in reallocation decisions—it only replans those queries that share streams with the new query. In addition, SQPR uses hosts to *relay* streams to make them available to other hosts. Although the focus of SQPR is initial query planning, it also supports adaptive query optimisation by replanning already-running queries after their resource requirements have changed.

The evaluation results show that SQPR performs better than a current state-of-the-art planner and a hand-crafted heuristic, especially in scenarios when resources are scarce. It scales well in terms of the number of queries and base streams, while showing low overhead in practice. Experimental results from a deployment on a cluster of machines as part of a prototype DSPS demonstrate the feasibility of SQPR in practice.

In summary, the main contributions of this paper are: (1) a unified formal optimisation model of query planning with query reuse in DSPSs (§III); (2) an efficient and scalable query planning approach based on solving a constrained optimisation problem (§IV); and (3) an evaluation of SQPR in simulation and in comparison to another approach in a DSPS deployment (§V). We discuss related work in §VI and finish with future work and conclusions (§VII).

II. Query Planning

In this section, we give the necessary background to query planning in DSPSs and explain our requirements. A DSPS manages a set of distributed stream processing hosts. It executes continuous queries that transform base streams to result streams, which are then delivered to clients. When a new query is submitted to the DSPS, a query planner must find a query plan that allocates resources to the new query before it is added. We assume the following high-level requirements for query planning:

- R1: Query planning should be scalable to support a large number of queries and resources.
- R2: Query plans should satisfy the resource needs of queries and enable them to execute with good performance.
- R3: Query plans should not prevent future queries from being satisfied.
- R4: Query planning should exploit overlap among queries by sharing resources in query plans.

A query planner must be able to support a DSPS that consists of a large number of hosts (R1). A common assumption today is that processing hosts are over-provisioned and therefore have sufficient resources to support new queries. This considerably simplifies the query planning problem because the planner can choose among many hosts that can provide adequate query performance (R2). With the virtualisation of physical hosts in data centres, this becomes wasteful.

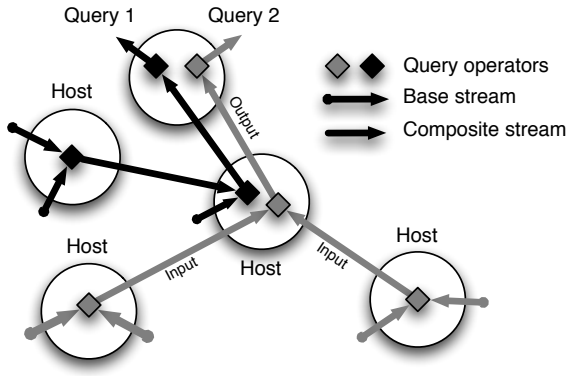


Fig. 1. DSPS with 5 hosts, executing two queries with operators and streams

Resources unused by a DSPS can be allocated to other applications in the same data centre; for example, computational resources such as CPU cores can be powered down to save energy. This has also implications on load-balancing of queries. While load-balancing of operators across hosts is in general desirable to achieve low latency in query processing, it may prevent resources from being reclaimed when the system serves only a low query workload.

Bad planning decisions may introduce resource bottlenecks at hosts, preventing resources from being exploited by future queries (R3). The implications of this for query planning is that it should perform well in scenarios with high contention for resources on certain hosts. Finally, the overlap between queries submitted by different users can be exploited to save resources through reuse (R4).

Next we introduce the system, query and resource models that are used in the rest of the paper.

A. System and query model

As shown in Fig. 1, we assume that a DSPS has hosts, streams and query operators. For ease of exposition, we assume that hosts, streams and query operators are time-invariant. We can therefore denote by $\mathcal{H} := \{1, \dots H\}$ the set of *hosts* and by $\mathcal{S} := \{1, \dots, S\}$ the set of all *data streams*. Streams may, for example, consist of relational tuples with a given schema [10]. Fig. 1 shows that each stream $s \in \mathcal{S}$ can either be a base or a composite stream. A *base stream* is a stream that is injected into the DSPS externally and that is available at a given host. We define $\mathcal{S}_h^0 \subset \mathcal{S}$ as the set of all base streams available at host $h \in \mathcal{H}$. A *composite stream* is a stream of result tuples that is generated by a query operator described below.

A stream $s \in \mathcal{S}$ has a given *average data rate*, denoted as $\varrho_s \in \mathbb{R}_+$. The data rate can be observed for base streams and estimated based on operator selectivities for composite streams. We assume that streams have a constant average data rate with a small variance.

Base and composite streams are transformed by *query operators* that continuously process tuples in a moving window. Examples of common relational streaming operators are join, select and project, although our model makes no assumptions

regarding specific semantics. The set of possible operators is denoted by $\mathcal{O} := \{1, \ldots, O\}$. An operator $o \in \mathcal{O}$ is described by a triplet $(\mathcal{S}_o, s_o, \gamma_o)$ where $\mathcal{S}_o \subset \mathcal{S}$ denotes the set of *input streams* that are transformed to a single *output stream* $s_o \in \mathcal{S}$ under *computational costs* $\gamma_o \in \mathbb{R}_+$.

Based on the model above, a DSPS must provide a *query plan* when a new query is submitted to the system. Intuitively, as illustrated in Fig. 1, a query plan is a mapping of query operators to hosts so that sufficient resources are available to support the processing of all associated data streams. We define the query planning problem more formally in §III-A.

B. Resource model

We consider three types of resources provided by hosts: (a) The amount of *computational resources* available at host h is denoted by $\zeta_h \in \mathbb{R}_+$. This could be a measure of the number of CPU cores on host h. (b) The maximum *outgoing host bandwidth* of host h amounts to $\beta_h \in \mathbb{R}_+$. This is a property of the network interface card of host h. (c) The available *network bandwidth* between hosts h and m is given by $\kappa_{hm} \in \mathbb{R}_+$. It is determined by the network topology in the data centres and properties of the network switches along the path. (For simplicity of presentation, we assume that final result streams of queries only need to be sent once to multiple clients—the same stream can be broadcast to multiple clients using a proxy host outside of the DSPS.)

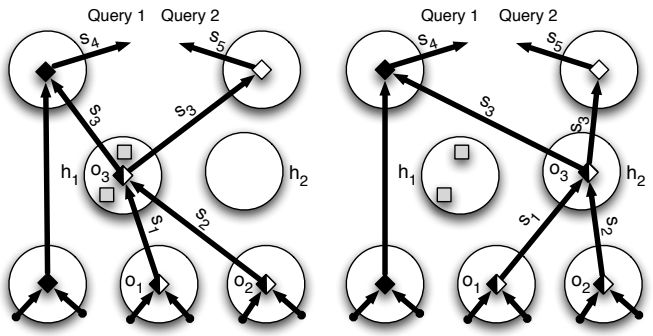
We have chosen these resources as they are the main resources consumed by a DSPS. They are also representative of different classes: computational resources and host bandwidth are examples of per-host resources, whereas network bandwidth depends on a pair of hosts. Other resources such as memory can be added in a similar way. Through operating system and network virtualisation, unused resources can be assigned to different applications outside of the DSPS, which results in a strong incentive for the DSPS to fully utilise its assigned resources.

We assume that operator $o \in \mathcal{O}$ consumes γ_o units of computational resources (e.g. as a percentage of time on a CPU core) when executed. The operator can only be assigned to a host that has sufficient computational resources available and sufficient host and network bandwidth to receive and send its input and output streams.

We assume that the DSPS has up-to-date information on the current resource utilisation of all hosts (cf. §IV-C). The resource consumption of existing operators can be observed. For operators of new queries, the resource consumption can be estimated based on operator semantics and the rates of the input streams using existing techniques [11]. We assume a simple cost model where the required processing resources for operators and the output stream network consumptions are linear functions of the rates of input streams.

C. Query planning with reuse

Overlap between sub-queries is common and is exploited in traditional DBMSs using views. The reuse of sub-queries in a DSPS reduces resource consumption—it takes advantage of already existing computation over streams and only consumes additional network resources to make result streams available to other queries. When there is a high degree of overlap between queries, this enables the DSPS to admit more queries.



(a) Reuse plans with two queries exploiting all resources on host h_1

(b) Reuse plans wasting CPU resources on network-bound host h_2

Fig. 2. Example of alternative query plans with reuse

In our model, queries can reuse streams of other queries. Fig. 2(a) gives an example of sub-query reuse with two queries 1 and 2. The two queries share stream s_3 on host h_1 produced by operator o_3. As a consequence, they also share operators o_1 and o_2. We define two streams to be *equivalent* if they are produced by the same operators using the same input streams. This makes the discovery of shared streams between queries simple by traversing their query plans. Note that this approach is only possible when the set of operators is well-known and operators are deterministic so that equivalences between operator instances can be established. For DSPSs that support user-defined operators [9], it is in general not possible to verify whether two sub-queries are equivalent.

Efficiently reusing shared sub-queries complicates planning when queries require more than one resource, such a CPU and network resources. Indeed, query reuse trades off network resources for CPU resources at a host. If a query planner greedily reuses sub-queries, it may exhaust the network resources before fully utilising CPU resources. This wastes resources in the system that become unusable to other queries.

Consider again the example given in Fig. 2(a). We assume that each host has the same amount of resources: it can support at most three query operators and support four large streams. In the allocation in Fig. 2(a), host h_1 executes operator o_3 and two other operators, which jointly use up its CPU resources. The four streams also deplete its network resources. (We assume that the streams for the other two operators have low data rates and can be ignored.) This leaves all resources of host h_2 available for future queries, potentially maximising the total number of supported queries.

An alternative, potentially less desirable, allocation is shown in Fig. 2(b). Here the shared operator o_3 has been allocated to host h_2 instead of h_1. Again, this almost entirely exhausts h_2's network resources, preventing further allocation of operators to it that require significant network resources. By creating a bottleneck in terms of network resources on host h_2, its spare

CPU resources are wasted to the system.

To work around this issue, we give the planner more freedom to reuse streams: we assume that hosts can *relay* streams to other hosts. By propagating a stream to another host with potentially more spare network resources, the planner can support more reuse with future queries. For example, in Fig. 2(b), a single stream s_3 could be relayed from h_2 to h_1 to remove the network bottleneck on h_2. The cost of relaying streams is that it complicates query planning even further by giving another degree of freedom to the query planner.

An interesting issue in the context of query planning is the need for load-balancing. While the plan in Fig. 2(a) does not load-balance operators and therefore may incur a higher processing latency of tuples processed by host h_1, it has the benefit that host h_2 remains idle and may be powered down. In contrast, the plan in Fig. 2(b) load-balances operators and may achieve lower processing latency. We believe that a query planner should provide control over the degree of load-balancing to the system administrator. For example, with a low query workload, load-balancing may be undesirable to achieve power efficiency but it may be more important with an almost fully-utilised system. As explained in §IV, SQPR provides control over load-balancing in planning decisions.

III. Optimisation Model

In our approach, we treat query admission, operator allocation and reuse as a single inter-related constrained optimisation problem. As described above, query reuse requires the re-planning of existing queries to remove bottlenecks in the system as new queries are added. This avoids prematurely exhausting specific resources of individual hosts.

However, the integration of operator allocation and reuse into a single model introduces a new challenge: it requires the query planner to explicitly account for the data streams that are sent between the hosts. We address this issue in the optimisation model by incorporating novel *acyclicity constraints*. These constraints ensure the absence of cycles in the query plan, which would correspond to acausal sequences of data streams. For example, an acausal sequence of streams arises if two hosts h_1 and h_2 send a base stream $s \in \mathcal{S}$ to each other, but neither host receives s from elsewhere. Acausal sequences of streams cannot be implemented in practise, and the optimisation model must ensure that any feasible solution is causal.

Next we formalise the query planning problem and describe the objective function and the various constraint sets of our optimisation problem.

A. Query planning problem

We define a *query plan* as a tree with node and arc labels. The node labels are of the form $\langle h, o\rangle$, where $h \in \mathcal{H}$ represents a host and $o \in \mathcal{O} \cup \{\mu\}$ denotes an operator (where μ is the *relay operator*, which enables hosts to exchange streams, as introduced in §II-C). The arc labels are of the form $\langle s\rangle$, where $s \in \mathcal{S}$ represents a data stream. Contrary to ordinary trees, the root node has an outgoing arc to send the result stream to the client, and each leaf node has one or more incoming arcs to receive data from stream sources.

We can interpret a query plan as follows. A node with label $\langle h, o\rangle$, $o \in \mathcal{O}$, states that host h executes operator o. Likewise, a node with label $\langle h, \mu\rangle$ means that host h *relays* a stream to other hosts that require them. An arc with label $\langle s\rangle$ represents a flow of stream s. A query plan for query $q \in \mathcal{S}$ needs to satisfy the following four conditions:

- C1: The arc emanating from the root node has label q.
- C2: For node $\langle h, o\rangle$, $o \in \mathcal{O}$, the labels of the incoming arcs form a superset of $\mathcal{S}_o$, while the label of the outgoing arc is s_o.
- C3: A node with label $\langle h, \mu\rangle$ has only one incoming arc. The label of that arc coincides with the label of the outgoing arc.
- C4: If arc $\langle s\rangle$ enters leaf node $\langle h, o\rangle$, then $s \in \mathcal{S}_h^0$.

Condition C1 ensures that the query plan matches the query. C2 and C3 ensure the correct behaviour of hosts that execute operators and relay streams, respectively. C4 requires the query to be satisfied from the repeated application of operators to base streams. We say that a query plan is *feasible* if the involved hosts and links have enough available resources to perform the assigned tasks.

B. Formal model

We assume that query planning has multiple objectives. The aim is to maximise the number of satisfied queries, minimise the usage of CPU and network resources, and possibly balance the load between the hosts of the network. A formal description of the query planning model relies on the following *decision variables*:

$$d \in \mathbb{B}^{H\times S}, \quad x \in \mathbb{B}^{H\times H\times S}, \quad y \in \mathbb{B}^{H\times S}, \qquad \text{(III.1)}$$
$$z \in \mathbb{B}^{H\times O}, \quad p \in \mathbb{R}_+^{H\times S}.$$

The binary variable d_{hs} indicates whether host $h \in \mathcal{H}$ provides stream $s \in \mathcal{S}$ ($d_{hs} = 1$) or not ($d_{hs} = 0$). To be able to provide stream s, host h must receive or generate s. This requirement is enforced by the variables x, y, and z. Our constraints ensure that $x_{hms} = 1$ if and only if host $h \in \mathcal{H}$ sends stream $s \in \mathcal{S}$ to host $m \in \mathcal{H}$. Likewise, we have $y_{hs} = 1$ if and only if stream $s \in \mathcal{S}$ is available at host $h \in \mathcal{H}$, and $z_{ho} = 1$ if and only if host $h \in \mathcal{H}$ executes the operator $o \in \mathcal{O}$. In the following, we call x_{hms} a *flow* variable, y_{hs} an *availability* variable, and z_{ho} an *operator* variable.

The continuous variable p_{hs} denotes the *potential* of stream $s \in \mathcal{S}$ at host $h \in \mathcal{H}$. Its meaning will become clear when we discuss the acyclicity constraints below. The tuple (d, x, y, z, p) uniquely identifies a *query plan*, as defined in the previous section. The constraints of our optimisation model ensure that only feasible query plans are considered.

Our query planning model has four objectives: maximise the number of satisfied queries (O_1), minimise the system-wide network usage (O_2), minimise the usage of computational resources (O_3), and potentially balance the load between

network hosts (O_4):

$$O_1 := \sum_{\substack{s\in\mathcal{S},\\ h\in\mathcal{H}}} d_{hs}\,, \qquad O_2 := \sum_{\substack{s\in\mathcal{S},\\ h,m\in\mathcal{H}}} \varrho_s x_{hms}\,,$$
$$O_3 := \sum_{\substack{h\in\mathcal{H},\\ o\in\mathcal{O}}} \gamma_o z_{ho}\,, \quad O_4 := \max_{h\in\mathcal{H}} \sum_{o\in\mathcal{O}} \gamma_o z_{ho}\,. \tag{III.2}$$

Since $d_{hs} = 1$ if and only if host h provides stream s, objective O_1 relates to the number of satisfied queries. Recall that ϱ_s represents the data rate of stream s, while $x_{hms} = 1$ if and only if host h sends stream s to host m. Hence, objective O_2 measures the system-wide network usage. Objective O_3 relates to the system-wide usage of CPU resources.

Objective O_4 measures the maximum resource consumption at any host. This objective allows us to balance the load between the networks hosts (if desirable): by minimising the maximum resource consumption over all hosts, O_4 penalises deviations of the resource consumption of any single host from the resource consumption of the other hosts. Note that strictly speaking, objective O_4 is not linear due to the maximum operator. However, one can use standard reformulation techniques to linearise this objective (cf. [12]).

Due to the conflicting nature of the different objectives, we cannot optimise all objectives simultaneously. Instead, we must seek Pareto efficient solutions with the property that no objective can be improved without deteriorating at least one of the other objectives. Such solutions are obtained by maximising a weighted sum

$$\lambda_1 O_1 - \lambda_2 O_2 - \lambda_3 O_3 - \lambda_4 O_4$$
$$\text{for some constants } \lambda_1, \lambda_2, \lambda_3, \lambda_4 \geq 0. \tag{III.3}$$

The parameters associated with the objective functions enable us to control the trade-off between the individual objectives. Exposing these parameters to SQPR allows our planner to control the degree of load-balancing in the query allocation (cf. §IV). For example, if we fix some values for (λ_1, λ_2) and set $(\lambda_3, \lambda_4) := (1, 0)$, we minimise the system-wide consumption of CPU resources. In contrast, we aim to obtain an even consumption of CPU resources across all hosts if we set $\lambda_3 := 0$ and $\lambda_4 := 1$. Intermediate settings, such as $(\lambda_3, \lambda_4) := (0.5, 0.5)$, allow us to trade off the conflicting goals of minimal resource consumption and load balancing.

A query plan, represented by the variables (III.1), cannot be chosen freely but must satisfy a number of technical and physical constraints in order to be feasible. For instance, a query plan must not require more resources than are available in the system. The necessary constraints are explained next:

Demand constraints. This constraint group involves the decision variable d. Host h can satisfy requests for stream s only if h possesses stream s and if s is actually demanded by users:

$$d_{hs} \leq \delta_s y_{hs} \quad \forall\, h \in \mathcal{H},\ s \in \mathcal{S}, \tag{III.4a}$$

where $\delta_s \in \mathbb{B}$ is an indicator variable that specifies whether stream s is requested ($\delta_s = 1$) or not ($\delta_s = 0$). Thus, d_{hs} can only have value 1 if $\delta_s = 1$ (stream s is required) and $y_{hs} = 1$ (host h possesses stream s).

To ensure the correctness of objective O_1, we restrict the queries considered in O_1 to those that are indeed requested:

$$\sum_{h\in\mathcal{H}} d_{hs} \leq \delta_s \quad \forall\, s \in \mathcal{S} \tag{III.4b}$$

If (III.4b) were missing, objective O_1 could account for streams that are not requested. Note that (III.4b) also implies that each requested data stream may only be served by at most one host. Indeed, if we allowed multiple hosts to serve one and the same request, then objective O_1 would no longer count the number of satisfied queries. However, our optimisation model can be readily adapted to systems, in which it is desirable to serve data streams by multiple hosts. In this case, objective O_1 would sum over new auxiliary variables q_s, where $q_s = 1$ if and only if $\sum_{h\in\mathcal{H}} d_{hs} > 0$, that is, if any host provides s.

Availability constraints. This constraint group relates the availability variable y to the flow variable x and operator variable z. Host m can possess stream s only if s is injected into m from some other host h, or if s is the output of a query operator executed at m, or if s is a base stream having its source at host m.

$$y_{ms} \leq \sum_{h\in\mathcal{H}} x_{hms} + \sum_{o\in\mathcal{O}:s_o=s} z_{mo} + \mathbf{1}_{[s\in\mathcal{S}_m^0]} \quad \forall\, m \in \mathcal{H},\ s \in \mathcal{S} \tag{III.5a}$$

Note that the variable y_{ms} on the left-hand side of (III.5) can only have the value 1 if $x_{hms} = 1$ for some $h \in \mathcal{H}$ (s is received from some other host h), if $z_{mo} = 1$ for a suitable operator $o \in \mathcal{O}$ (s is generated by host m), or if $s \in \mathcal{S}_m^0$ (s is a base stream that is available at host m). In that case, $y_{ms} = 1$ signalises that stream s is available at host m.

Host h can apply operator o only if all corresponding input streams $s \in \mathcal{S}_o$ are available at h.

$$z_{ho} \leq y_{hs} \quad \forall\, h \in \mathcal{H},\ o \in \mathcal{O},\ s \in \mathcal{S}_o \tag{III.5b}$$

This constraint implies that $z_{ho} = 0$ (host h does not apply operator o) as soon as $y_{hs} = 0$ for some input stream s of operator o. Finally, host h can transmit stream s to host m only if s is available at h.

$$x_{hms} \leq y_{hs} \quad \forall\, h, m \in \mathcal{H},\ s \in \mathcal{S} \tag{III.5c}$$

Resource constraints. The joint data rate of all streams transmitted from host h to host m may not exceed the available network bandwidth.

$$\sum_{s\in\mathcal{S}} \varrho_s x_{hms} \leq \kappa_{hm} \quad \forall\, h, m \in \mathcal{H} \tag{III.6a}$$

The joint data rate of all streams flowing into host m may not exceed its incoming host bandwidth.

$$\sum_{s\in\mathcal{S}} \sum_{h\in\mathcal{H}} \varrho_c x_{hms} \leq \beta_m \quad \forall\, m \in \mathcal{H} \tag{III.6b}$$

The joint data rate of all streams emitted from host h may not

exceed its outgoing host bandwidth.

$$\sum_{s\in\mathcal{S}}\sum_{m\in\mathcal{H}} \varrho_s x_{hms} + \sum_{s\in\mathcal{S}} \varrho_s d_{hs} \leq \beta_h \quad \forall\, h \in \mathcal{H} \qquad \text{(III.6c)}$$

Note that h can transmit stream s to another host m or to a client that requests s. For simplicity, we assume that a host has to transmit the same result stream only once to clients because a proxy host distributes the result stream among multiple clients. This assumption is not restrictive and can be lifted by a straightforward modification of our optimisation model.

The CPU resources consumed at host h to carry out query operators $o \in \mathcal{O}$ may not exceed the available budget.

$$\sum_{o\in\mathcal{O}} \gamma_o z_{ho} \leq \zeta_h \quad \forall\, h \in \mathcal{H} \qquad \text{(III.6d)}$$

Acyclicity constraints. We must ensure that any data stream available at a host has a real source. Without any further constraints, however, streams could arise from self-sustaining feedback loops. The reason for this nonsensical effect is that streams can be duplicated freely anywhere in the system and sent to multiple destinations. We now assign to each stream s and host h a potential or "elevation" p_{hs}. In order to avoid cycles, we require stream s to flow "downhill" with respect to its potential, that is, s can flow from h to m only if the potential p_{hs} exceeds p_{ms} by at least 1.

$$p_{hs} \geq p_{ms} + 1 - M(1 - x_{hms}) \quad \forall\, h, m \in \mathcal{H},\ s \in \mathcal{S} \qquad \text{(III.7)}$$

The constant M may be set to any value larger than $|\mathcal{H}| + 1$ implying that (III.7) does not restrict the choice of p if $x_{hms} = 0$. Note that the acyclicity constraints (III.7) become necessary because we integrate the query reuse and operator placement problem. To the best of our knowledge, these two problems have never been considered jointly by a single optimisation model, which explains why previous DSPS optimisation models did not contain acyclicity constraints.

The query planning problem can thus be formulated as a mixed integer linear program (MILP), solvable by standard branch and bound algorithms [12]:

$$\begin{aligned} \underset{d,x,y,z,p}{\text{maximise}} \quad & \lambda_1 O_1(d) - \lambda_2 O_2(x) - \lambda_3 O_3(z) - \lambda_4 O_4(z) \\ \text{subject to} \quad & \text{(III.4)–(III.7).} \end{aligned} \qquad \text{(III.8)}$$

IV. SQPR Planner

The optimisation model from the previous section is *static*: it assumes that all queries are known and that the query planning problem only needs to be solved once. In practise, however, query planning is highly *dynamic*: queries arrive continuously over time, the resource requirements of queries may change, and the query planning problem needs to be solved repeatedly.

Another problem arises due to the size of the optimisation problem: when a new query is added, the query planner would have to re-plan all existing queries from scratch to obtain an optimal set of plans. As we prove in the Appendix, our formulation of the stream query planning problem is strongly $\mathcal{NP}$-hard, rendering its solution intractable for large problems.

Instead, we search for an approximate solution that restricts re-planing to a subset of queries, which are "related" to the new queries added to the system: SQPR only reconsiders the allocation of those operators that share base or composite streams with the new query to be added. This provides a good trade-off in that it re-allocates operators that may be involved in query reuse without touching operators that are independent from the new query. The cost of query planning remains independent of the total number of queries and base streams in the system, achieving a more scalable solution.

In this section, we show how to incorporate the optimisation model into our SQPR planner to meet the requirements of real-life DSPSs. To this end, we start with a discussion of initial resource allocation for new queries based on cost estimates. We then consider adaptive re-planning of queries based on observed resource consumption. We finish with a description of the architecture of SQPR.

Algorithm 1 SQPR: Initial-Query-Planning

```
1: initial solution: set (d, x, y, z, p) := 0
2: wait for a new query q ∈ S
3: if d_hq = 0 for all h ∈ H then
4:     fix optimisation variables relating to irrelevant streams
5:     solve optimisation model (III.8) with constraint (IV.9)
6:     update solution (d, x, y, z, p) if successful
7: for h ∈ H do
8:     if ({x_hms}_{m,s}, {z_ho}_o) has changed then
9:         notify host h of changed streams and operators
```

A. Initial query planning

For each new query to be admitted, SQPR solves a variant of problem (III.8). This variant ensures that the new solution does not drop already admitted queries. Also, all decision variables that are not directly related to the new query are fixed in order to reduce the size of the optimisation model.

The initial query planning process of SQPR is summarised in Algorithm 1. Before any queries have been submitted, SQPR starts with the initial solution $(d, x, y, z, p) := 0$ to the optimisation problem (III.8), i.e. no streams exist in the network, and no operators are assigned to hosts (line 1). When a new query $q \in \mathcal{S}$ arrives (line 2), SQPR first checks whether the same query q has already been admitted: this is the case if $d_{hq} = 1$ for some host $h \in \mathcal{H}$ (line 3).

If the query q has not yet been admitted, SQPR solves the optimisation problem (III.8) with the additional constraint that

$$\sum_{h\in\mathcal{H}} d_{hs} = 1 \qquad \text{(IV.9)}$$

for each already existing stream $s \in \mathcal{S}$. Constraint (IV.9) ensures that the new solution does not drop already admitted queries. It does allow, however, that already admitted queries are allocated to different hosts. In practise, this flexibility can

lead to better solutions, but it does require that operators are potentially migrated between hosts.

Remember that the parameters $\lambda_1, \ldots, \lambda_4$ in (III.8) allow SQPR to favour solutions that minimise the consumed resources (large values for λ_2 and λ_3) or to focus on solutions that balance the load between the various hosts (large value for λ_4). The choice of $\lambda_1, \ldots, \lambda_4$ depends on the environment, in which SQPR is run, and the weights can be chosen statically or adapted dynamically depending on the workload. In our experiments in §V, we choose $\lambda_1 := M$, where M is a sufficiently large number. This ensures that the number of admitted queries (objective O_1) is the most important objective. The second weight is $\lambda_2 := 1/\sum_{h \in \mathcal{H}} \beta_h$, which scales the system-wide network usage (objective O_2) to a number between 0 and 1. The third weight is $\lambda_3 := 1/\sum_{h,m \in \mathcal{H}} \kappa_{hm}$ that scales the aggregated usage of CPU resources to a number between 0 and 1. Finally, the fourth weight is set to $\lambda_4 := 1$ and ensures that the usage of CPU resources on the most heavily used host (objective O_4) receives the same weight as the average consumption of CPU resources (objective O_3). We thus try to strike a balance between a minimal resource consumption and an even load distribution.

To avoid solving the full optimisation problem, SQPR solves smaller subproblems, in which some of the decision variables relating to "irrelevant" streams and operators are fixed. More precisely, let $\mathcal{S}(q) \subset \mathcal{S}$ denote the set of all data streams that can appear in query plans for q. Likewise, let $\mathcal{O}(q) \subset \mathcal{O}$ denote the set of all query operators that can appear in query plans for q. Both sets can be determined recursively. To speed up the optimisation, decision variables in (III.8) that correspond to streams $s \notin \mathcal{S}(q)$ and operators $o \notin \mathcal{O}(q)$ are fixed to their value in the previous solution (line 4). This means that SQPR optimises only over those streams and operators that are related to the new query q, while excluding the possibility of re-planning other queries and operators.

As an example, consider Figure 2(a) and assume that stream s_4 is already satisfied, while stream s_5 has just entered the system. Our fixations imply that SQPR would optimise over streams s_1, s_2, s_3 and s_5 and all three operators, while fixing all decision variables corresponding to stream s_4. Note that the solution to our pruned optimisation model is in general suboptimal, and there is no non-trivial a priori bound on the incurred losses in optimality. However, we can bound the incurred losses a posteriori by solving a simpler "optimistic" variant of model (III.8) that disregards the bandwidth constraints (§V). Our experiments in §V indicate that the losses due to our variable fixations are below 25% in practise.

Fixing unrelated decisions reduces the number of streams and operators in the query planning problem from S and O to $|\mathcal{S}(q)|$ and $|\mathcal{O}(q)|$, respectively. Depending on the complexity of the operators, we have that $|\mathcal{S}(q)| \ll S$ and $|\mathcal{O}(q)| \ll O$. Thus, even though the problem remains $\mathcal{NP}$-hard, we have managed to drastically reduce its size. Note that this does not affect the number of hosts considered and this remains a critical parameter in our problem.

Even with this simplification, it may be unrealistic to solve problem (III.8) to optimality. In practice, one can prematurely terminate the branch and bound algorithm after a given time interval and use the best solution that the method found [12].

If query planning is successful within the given time limit, the solution (d, x, y, z, p) is updated (line 6). Finally, hosts are notified of any changes in the assignment of queries (line 9).

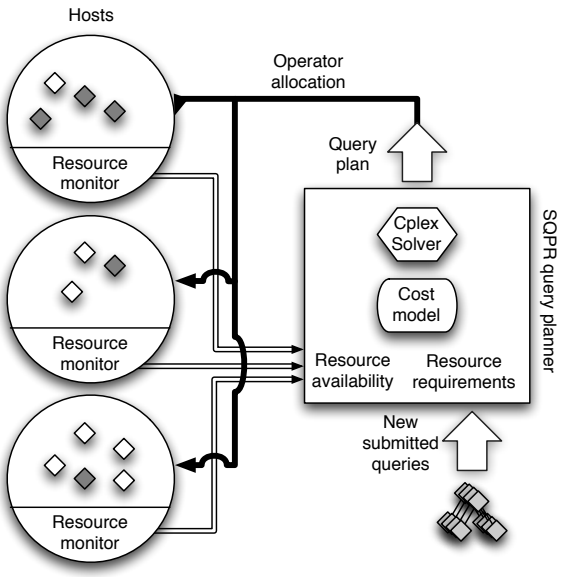


Fig. 3. DSPS architecture with SQPR query planner

B. Adaptive query planning

The initial query planning above is based on a cost model for estimating resource consumption, which may give inaccurate results. In addition, the resource consumption of queries may change as data rates and tuple distributions of base streams change. As a consequence, SQPR must adaptively revisit past planning decisions when the actual resource consumption of queries deviates from initial estimates.

We support this in SQPR by (conceptually) removing and re-adding queries to the DSPS that require re-planning. SQPR stores the resource estimates used during initial planning. It then monitors the resource consumption of all queries and periodically constructs a list of queries (a) for which the resource consumption differs from the initial estimates by a given threshold or (b) that suffer from a shortage of resources on a host. It then re-plans these queries by considering the system without those queries and re-adding them. Based on these new query plans, the DSPS migrates operators between hosts to reflect the plans, as done in other systems [4].

C. System architecture

The architecture of a DSPS that uses SQPR is shown in Fig. 3. We implemented SQPR as part of the experimental Dependable Internet-Scale Stream Processing (DISSP) system developed in our group. The SQPR query planner is a separate centralised component that controls resource allocation on DISSP hosts. It interacts with DISSP hosts to get resource information and instructs them to instantiate new operators

based on admitted query plans. We next describe the main system components in more detail.

DISSP system. The DISSP system is designed to provide a scalable and dependable stream processing service across a large number of hosts. It is written in Java and follows a relational streaming model with a set of common streaming operators including join, filter and project. Queries are executed by running operators across a set of DISSP hosts. DISSP hosts exploit multiple CPU cores by scheduling operators using a pool of worker threads. Data streams between engines are exchanged through TCP connections. DISSP hosts are controlled using a management interface for monitoring resources and handling operators. Base streams are added through external processes that produce a stream of tuples based on an agreed relational schema.

SQPR query planner. Our planner is implemented as a standalone server. It receives up-to-date information from DISSP hosts on running operators. When new queries are submitted, it estimates the resource requirements of queries based on our basic cost model from §II-B. Query planning is then expressed as an optimisation problem and passed to the CPLEX 11.2 optimisation software [12]. CPLEX is a state-of-the-art solver for (primarily) linear and mixed-integer linear programs. In our experiments, we use the standard parameters suggested by CPLEX. The solver is invoked with a fixed timeout (as reported in §V), after which the query is not admitted if no solution was found. If a feasible query plan is discovered, it is instantiated by SQPR using the management interface of DISSP hosts.

Resource monitoring. To make our query planning approach compatible with the DISSP system, we provide an additional capability of reporting resource utilisation to SQPR. Each DISSP host runs a resource monitor that periodically samples the utilisation of all CPU cores and the used network resources. This information when reported to SQPR provides an up-to-date view of the current system state.

V. Evaluation

Our evaluation goals are to investigate the planning efficiency, scalability, and overhead of SQPR initial query planning. By reusing streams and placing operators in a non-myopic way, SQPR should operate in a resource-efficient manner. For a practical deployment, it must scale to a large number of hosts and resources, as well as complex queries. Finally, SQPR should plan new queries within an acceptable time limit. While we are willing to sacrifice global optimality, the incurred optimality gap should be reasonably small.

In our evaluation, we use simulation to investigate query planning in larger systems and a real-world deployment of SQPR as part of our prototype DISSP system on the Emulab cluster testbed [13].

For the query workload, we consider a large set of base streams that are uniformly distributed over the hosts. We randomly create 1,000 queries that consist in equal parts of two-way, three-way and four-way joins over the base streams. Joins have a selectivity in the range of 0.1%–0.5%. The base streams in a query are chosen according to a Zipfian distribution with parameter 1. This guarantees a certain amount of overlap between queries.

A. Simulation

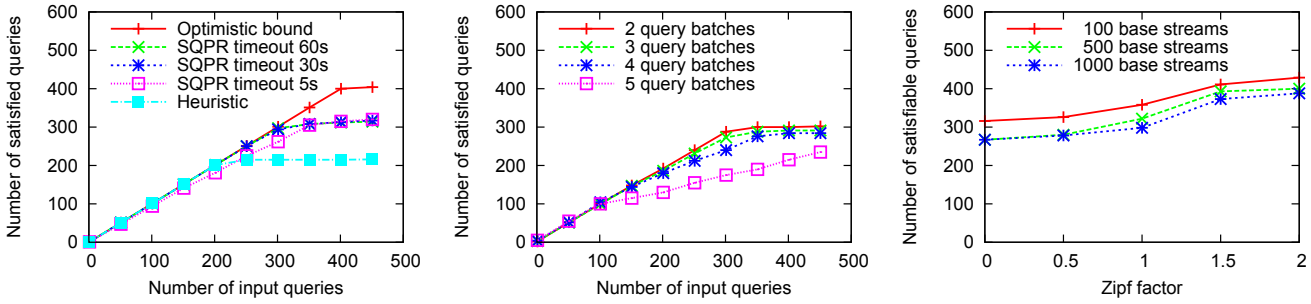
First we simulate a system with 50 hosts using a custom-built simulator. We consider 500 base streams and assume that the average data rate of each is 10 Mbps. All network links between hosts have a capacity of 1 Gbps. We set the available CPU resources on hosts in such a way as to obtain a CPU- and bandwidth-constrained environment with our query workload.

In addition to query planning using SQPR, we also compare to the following two query planning strategies in our simulator:

Heuristic planner. We compare SQPR to a heuristic planner with query reuse that is inspired by existing approaches [14]. Whenever a new query $q \in \mathcal{S}$ is submitted, the heuristic planner generates all abstract query plans (i.e. query plans whose operators are not yet assigned to hosts) for q. Depending on the operator semantics, the number of abstract query plans can be exponential in size of the query. Since we only consider queries that result from two-way, three-way and four-way joins in our experiments, however, an exhaustive enumeration of all abstract query plans is feasible. For each abstract query plan, the planner iterates over all hosts $h \in \mathcal{H}$ and tries to implement the abstract query plan at host h. To this end, it ensures that host h receives all of the required data streams $s \in \mathcal{S}$ from other hosts that have s. In this step, the planner tries to aggressively exploit sub-query reuse by favouring the transfer of complete sub-queries over base streams that require the application of more operators within h. Each candidate placement that results in a feasible query plan is evaluated according to the weighted objective function explained in §III-B, and the best placement is chosen.

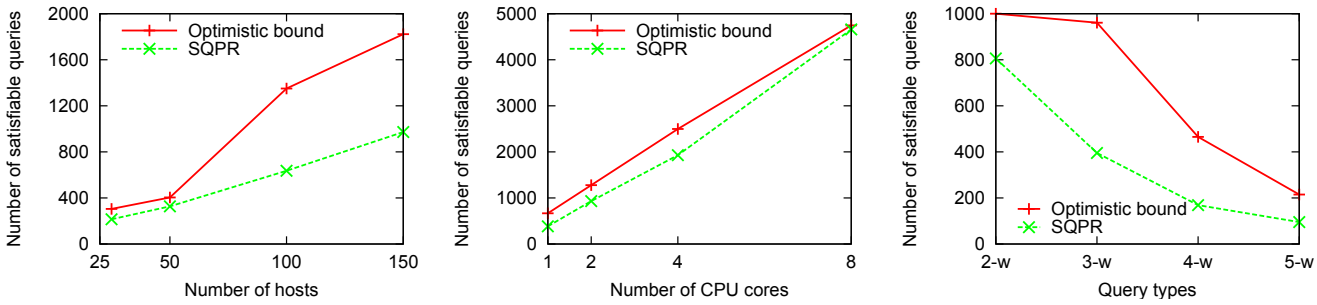
Optimistic bound. We also compare SQPR to an optimistic upper bound to estimate how close SQPR's solution is to the optimum. Since an optimal planner is not implementable, we provide an upper bound by aggregating all hosts into a single "aggregate host". This aggregate host has all base streams, and its CPU resources equal the sum of all CPU resources in the system. Since the aggregate host is the only host in this synthetic network, we can omit all constraints relating to the network. In this case the optimisation model (III.8) simplifies dramatically and allows for an analytical solution. The amount of queries satisfiable by the aggregate host constitutes an upper bound on the number of queries satisfiable by our DSPS, even if all optimisation problems were solved to global optimality and if we did not decompose problem (III.8). By construction, the true optimal solution is below this optimistic upper bound.

1) Planning efficiency: In our first experiment, we investigate the number of queries that SQPR manages to allocate successfully. We simulate the submission of one query at the time and observe if it can be admitted. Fig. 4(a) shows the relation between submitted and satisfied queries for SQPR under different timeouts (5 secs, 30 secs and 60 secs) in



(a) Planning efficiency (b) Efficiency with batching (c) Efficiency with overlap

Fig. 4. Efficiency of query planning



(a) Scalability in terms of hosts (b) Scalability in terms of resources (c) Scalability in terms of query complexity

Fig. 5. Scalability of query planning

comparison with the heuristic planner and the optimistic bound.

All three approaches accept new queries until the system saturates. SQPR outperforms the heuristic planner, even though the latter exploits overlap, considers various query placements and employs the same objective function. This is because the heuristic planner neither reconsiders previous allocation decisions nor distributes query plans over multiple hosts. From the optimistic bound, we deduce that the optimality gap of SQPR is less than 25%.

Given that we were unable to solve the full-scale query planning problem within an hour, we conclude that the problem reduction technique from §IV-A leads to an acceptable trade-off between optimality and efficiency. Unless stated otherwise, SQPR uses a timeout of 30 secs in all of the following results, which we consider an acceptable upper bound when users submit new queries to the system.

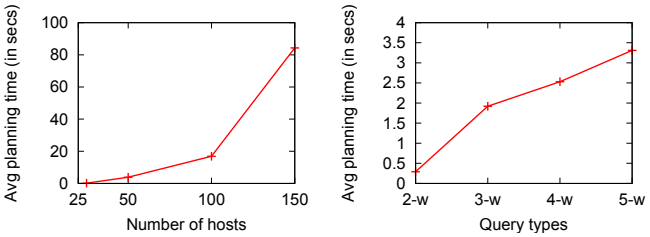
Instead of submitting each new query individually, we explore the impact of batching multiple queries at submission time on planning efficiency in Fig. 4(b). SQPR provides query plans for each batch of n queries with a timeout of $30n$ secs. As the result shows, this reduces the number of satisfied queries because it decreases the possibilities for problem reduction. It leads to difficult optimisation problems that cannot be solved within the employed time limits. We conclude that batching queries to large groups is not desirable in practice.

The impact of the degree of overlap between queries on planning efficiency is shown in Fig. 4(c). We vary the number of base streams, as well as the Zipf distribution according to which base streams are selected in queries to control overlap. Setting the Zipf parameter to zero implies that base streams are chosen uniformly. Note that for the same Zipf distribution, a smaller number of base streams leads to higher overlap. As expected, the performance of SQPR increases with the degree of overlap. This is caused by the fact that SQPR exploits query reuse whenever it is beneficial.

2) Scalability: To evaluate the scalability of SQPR, we investigate the impact of additional resources (in the form of more or more powerful hosts) and more complex query operators. Fig. 5(a) shows that the performance of SQPR grows super-linearly in the number of hosts. Nevertheless, the gap between the optimistic bound and SQPR widens as the number of hosts increases. As we will show later on, this is due to the fact that our optimisation model does not scale well in the number of hosts. Larger number of hosts imply larger optimisation models, which makes it increasingly difficult to find well-performing query plans within the set timeout.

Fig. 5(b) shows that SQPR scales well in the available resources. In this experiment, we increase all available network bandwidths from 1 Gbps to 10 Gbps and consider hosts with more CPU cores. Due to our decomposition technique, the size of SQPR's optimisation model does not increase in the available CPU resources, resulting in near optimal solutions.

The impact of the query complexity is shown in Fig. 5(c). Here we assume that all submitted queries are 2- to 5-way



(a) Planning time vs. host number (b) Planning time vs. query type

Fig. 6. Running times of query planner

joins. More complex queries require more resources, reducing the number of queries that SQPR can admit. From the graph, we conclude that the efficiency of SQPR (compared to the optimistic bound) is more or less independent of the query complexity. We will see below that this is due to the fact that the size of our optimisation model is more sensitive to the number of hosts than to the complexity of submitted queries.

In summary, SQPR scales well with the amount of available computational resources and the query complexity, but its performance deteriorates for large networks. As we will now show, this is due to the size of the optimisation problems.

3) Overhead: Next we investigate the impact of the number of hosts and the query complexity on the solution time of SQPR. We set the timeout to 100 secs and record the time required to plan queries when 75%–95% of resources are consumed. We expect planning to be most challenging when there are few resources left (i.e. the potential infeasibility cannot be easily detected), but the satisfaction of further queries requires restructuring among many hosts. In the following, a runtime below 100 secs implies that the reduced optimisation problem has been solved to optimality by SQPR.

Fig. 6(a) presents average planning time for different network sizes. The results show that SQPR can solve problems with up to 100 hosts to optimality, while larger systems require significantly longer. In most of the cases, SQPR is still able to provide a feasible solution.

Fig. 6(b) illustrates the impact of the query complexity on solution time with 50 hosts. As expected, complex queries decrease the potential of problem reduction and therefore lead to more challenging optimisation problems. Note, however, that the increase in runtime is smaller than in Fig. 6(a).

Overall we conclude that SQPR's planning time scales well in the query complexity, but is sensitive to the number of network hosts. In future work, we plan to address this issue by using a hierarchical approach that optimises over sets of hosts, each of which is optimised independently.

B. Cluster deployment

Next we investigate the performance of SQPR as part of a deployment of our prototype DISSP system on a cluster of machines. We ran our system on 15 hosts on the Emulab testbed [13] with a 10 Mbps LAN. Each host is a PC with 2 GB of RAM and a 3 GHz CPU. We gradually increase the number of input queries by periodically submitting 50 queries to the query planner and then deploying the satisfied queries. The queries are 2- and 3-way joins over 300 base streams with 10 Kbps output rates distributed uniformly across hosts. Using off-line measurements, we determined that each host could support up to 15 2- and 3-way joins before it reached CPU saturation. We used this to seed the cost model of the query planners.

To compare the performance of SQPR, we use SODA [9], a state-of-the-art query planner and runtime optimiser that is part of IBM's System S stream processing platform [1]:

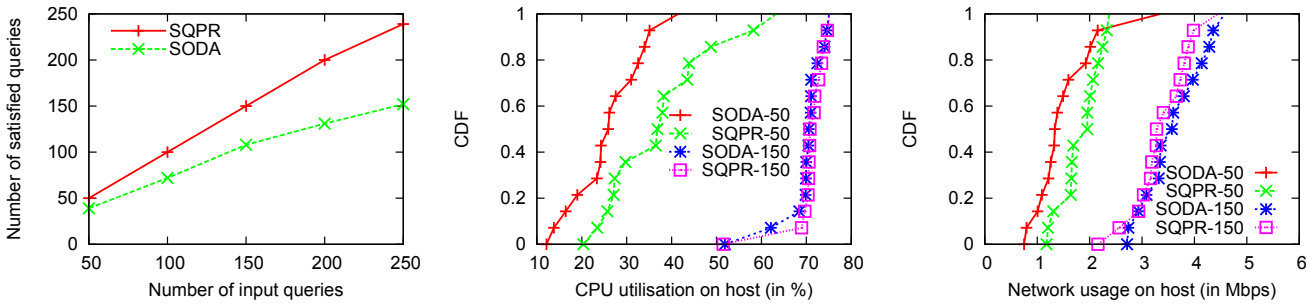
SODA considers queries that support flexible resource allocations reflecting different quality-of-service (QoS). The goal of SODA is to admit a set of queries that maximise the total QoS of the DSPS while obeying a user-defined query ranking. SODA first admits queries based on their overall resource consumption and system availability. For admitted queries, it solves the resource allocation problem using a mixture of constrained optimisation models and heuristics. SODA also performs runtime adaptation to workload variations. Similar to SQPR, SODA does query planning in epochs.

SODA exploits stream reuse by *gluing* together different query *templates*. Templates are user-defined and describe the query structure. To minimise network usage among hosts, query operators first seek to use input streams from the local host. If this is not possible, the input streams are received once from the original host and locally propagated to other co-located operators.

In contrast to SODA's multi-stage approach to query planning, SQPR solves the combined problem of query admission and operator placement in one step. A query can only be admitted after a feasible placement of its operators is found. SQPR also performs advanced planning decisions based on stream reuse by dynamically changing the query plan and relaying streams across hosts. In contrast, the SODA scheduler is bound by the initial user-given query plan and, once admitted, has to follow its structure for all subsequent epochs. Finally, SODA performs query planning across all queries, whereas, SQPR considers only the new query and those already admitted that share streams with the new query.

We have implemented the basic functionality of SODA, namely query admission (the *macroQ* stage of the SODA scheduler), operator placement, and resource allocation [9]. The last two steps are solved by a combination of optimisation (*macroW* stage) and heuristic techniques (*miniW* stage). The SODA heuristic has a dual role: if a solution is not found within the time limit by the *macroW* stage, it provides the final operator placement. However, if a solution is found, it further attempts to improve it according to the objective function by exploring different local operator swaps. Since it provides the final operator placement in both cases, we have decided to use the *miniW* stage.

To compare SODA with SQPR, we assign queries with specific resource requirements. As a result, SODA's stages of resource allocation tuning, which adjust operator resource allocation to minimise the load-balancing objective, are not applicable—SQPR satisfies fixed resource demands and thus



(a) Planning efficiency (b) Distribution of CPU resources (c) Distribution of network resources

Fig. 7. Results from DISSP cluster deployment with SQPR and SODA

cannot adjust resource requirements of queries. To maximise reuse opportunities in SODA, we combine query templates in a way that each stream is generated once and used by all other queries when needed. Finally, the objective function for the next experiments is set to load balancing (cf. IV-A).

1) Planning efficiency: Fig. 7(a) shows the number of admitted queries by the SQPR and SODA planners for every 50 input queries in the deployed system. Both planners are given the same set of input queries. Each query has a 30 secs planning timeout. SQPR accepts queries linearly, as long as the system is not too overloaded. As input queries arrive, SQPR accepts more queries than the SODA scheduler. At the last stage, the system reaches saturation. SQPR does not admit all 250 queries because no suitable placement for the operators is found within the timeout.

As the system approaches saturation, SODA is less flexible when placing queries on resource-constrained hosts. In addition, SODA is bound to match a fixed query template to a given host and resource layout. In cases of saturation, this limits flexibility and therefore decreases the admission rate of queries. Although SODA exploits reuse, it does this by gluing together many small queries to create large ones. Large queries, however, make the placement problem harder. In contrast, SQPR is able to adjust the query structure by relaying streams across hosts and is therefore able to find a placement even under tight resources conditions.

2) Resource distribution: Fig. 7(b) shows the distribution of CPU utilisation across hosts as a CDF plot in the cases of 50 and 150 input queries deployed by SQPR and SODA, respectively. These choices of input queries result in an under-utilised system and one that is close to overload. The figure shows that both planners balance the load across the hosts. Since SQPR admits more queries, it consumes more resources, as shown by the SQPR-50 line in the figure. When both SODA and SQPR accept a high number of queries (lines SQPR-150 and SODA-150), the CPU load is evenly distributed across hosts and the system is approaching overload.

Fig. 7(c) shows the distribution of network usage as the sum of sent and received data per host. We plot this for 50 and 150 submitted queries. The results show that both planners roughly manage to balance the network usage. For SQPR, this is achieved by the load-balancing objective of the planner.

3) Conclusions: The query planning decisions of SQPR assume fixed resource demands per query and the potential for extensive reuse between queries. This is exploited by relaying streams between hosts and reusing streams between operators. In this way, SQPR explores different possible query plans and chooses the best according to its objective function. When compared to SODA, it satisfies more queries because SODA assumes a fixed query template, which limits its search space for feasible placements. In addition, SODA struggles finding operator placements when it does not have the flexibility to adapt the resource requirements of admitted queries. For admitted queries, both planners are able to evenly balance the load across hosts and saturate the system.

VI. Related Work

Different aspects related to query planning, optimisation, and reuse have been studied in research projects such as STREAM [10], Gigascope [2] and Borealis [4].

Query planning. Past proposals for query planning in DSPSs assume an abundance of resources and leverage knowledge of topology and network bandwidth to deploy query operators efficiently [8]. Ahmad et al. [14] suggest an allocation technique that deploys operators at source hosts. Pietzuch et al. [15] minimise global network usage of operators based on optimisation in a metric space. In work on plan-based composite event detection [16], the authors focus on the efficient allocation of operators to minimise network usage. Lakshmanan [17] propose a biologically-inspired system to place operators on hosts while minimising end-to-end latency. In contrast, we consider multiple constraints on both computational and network resource when planning queries together with query reuse.

The closest work to ours is SODA [9], which performs query planning and runtime adaptation. We describe and compare to SODA's initial query planning in §V-B.

Query optimisation. In addition to initial query planning, researchers have investigated dynamic query optimisation at runtime, often with a focus on load-balancing and shedding. Markl et al. [11] propose that a query optimiser compares run-time statistics with the initial estimates and changes the query plan if necessary. The Borealis stream processing engine balances load by considering load correlations between

operators and moving operators to maintain low processing latency [6]. An approach for dynamic operators placement presented by Zhou et al. [18] places operators initially to reduce communication cost and then carries out dynamic load-balancing. In contrast, we are concerned with efficient initial query planning and admissions control—any of the above approaches can be applied to provide dynamic load-balancing at a finer granularity than SQPR.

Tatbul et al. introduce optimisation metrics for load-shedding [19] when the system is overloaded and model it as a linear program [20]. Feng et al. [21] address the combined load-shedding and resource allocation problem in a shared distributed stream processing platform with static placement of operators. They use centralised optimisation techniques and back-pressure to control the flow of data along the operators. Our work is complementary in that we attempt to do admissions control without having to shed data. If the workload changes at runtime, we can use these existing approaches for load-shedding to handle the overload condition until adaptively re-planning queries (cf. §IV-B).

Query reuse. Several efforts investigate the potential of query reuse in DSPS. The work on Synergy [22] provides a set of distributed algorithms to discover and exploit stream and operator reuse at deployment time. Similarly, Zhou et al. [23] leverage publish/subscribe communication to find common data interests across queries. XFlow [24] allows users to search already running queries for reuse opportunities. Multiple queries are optimised in the work by Seshadri et al. [25] by solving an approximation of the query optimisation problem exploiting operator re-use in hierarchical networks. We address the general problem of query planning in any network configuration by solving a formal optimisation problem.

VII. Conclusions

We presented SQPR, an optimisation-based query planner for DSPSs. SQPR is designed to perform well in data centres environment that are resource constraint, which makes initial query planning for new queries challenging. SQPR carries out query admission, allocation and reuse by solving a reduced optimisation problem. Our evaluation results show that SQPR is more resource efficient than a hand-crafted heuristic planner. It can also admits a larger number of queries than a state-of-the-art query planner. It scales well to increasing number of queries and resources without sacrificing planning efficiency.

In future work, we plan to extend our SQPR planner with support for more resources (including memory) and advanced cost models for predicting resource consumption. We also want to combine heuristics with SQPR to increase satisfied queries. To improve scalability in terms of hosts, we will explore a hierarchical decomposition of the problem. This would enable query planning across federated data centres by first assigning queries to sites and then planning queries within sites. Finally, we want to investigate how to decentralise the planning itself by having multiple SQPR instances responsible for different parts of a large-scale DSPS.

VIII. Acknowledgements

The authors would like to thank Marco Fiscato for his work on developing the prototype DISSP system.

References

[1] B. Gedik, H. Andrade, K.-L. Wu, P. S. Yu, and M. Doo, "SPADE: The System S Declarative Stream Processing Engine," in *SIGMOD'08*. ACM, 2008, pp. 1123–1134.

[2] C. Cranor, T. Johnson, O. Spatscheck, and V. Shkapenyuk, "Gigascope: A Stream Database for Network Applications," in *SIGMOD*, 2003.

[3] O. Cooper, A. Edakkunni, M. J. Franklin, W. Hong, S. R. Jeffery, S. Krishnamurthy, F. Reiss, and E. Wu, "HiFi: A Unified Architecture for High Fan-in Systems," in *VLDB'04*, August 2004.

[4] D. J. Abadi, Y. Ahmad, M. Balazinska, U. Çetintemel *et al.*, "The Design of the Borealis Stream Processing Engine," in *CIDR*, 2005.

[5] L. Chen, K. Reddy, and G. Agrawal, "GATES: A Grid-Based Middleware for Processing Distributed Data Streams," in *HPDC*, 2004.

[6] Y. Xing, S. B. Zdonik, and J.-H. Hwang, "Dynamic Load Distribution in the Borealis Stream Processor," in *ICDE*, 2005.

[7] P. Roy, S. Seshadri, S. Sudarshan, and S. Bhobe, "Efficient and Extensible Algorithms for Multi Query Optimization," in *SIGMOD'00*, 2000.

[8] G. T. Lakshmanan, Y. Li, and R. Strom, "Placement Strategies for Internet-scale Data Stream Systems," *IEEE Internet Computing, Special Issue on Data Streams*, vol. 12, no. 6, 2008.

[9] J. Wolf, N. Bansal, K. Hildrum, S. Parekh *et al.*, "SODA: An Optimizing Scheduler for Large-scale Stream-based Distributed Computer Systems," in *Middleware*, 2008.

[10] A. Arasu, B. Babu, and J. Widom, "The CQL Continuous Query Language: Semantic Foundations and Query Execution," *VLDB Jounal*, vol. 15, no. 2, pp. 121–142, 2006.

[11] V. Markl, V. Raman, D. E. Simmen, G. M. Lohman, and H. Pirahesh, "Robust Query Processing through Progressive Optimization," in *SIGMOD*, 2004.

[12] IBM, "ILOG CPLEX," www.ibm.com, 2010.

[13] B. White, J. Lepreau, L. Stoller, R. Ricci, S. Guruprasad, M. Newbold, M. Hibler, C. Barb, and A. Joglekar, "An Integrated Experimental Environment for Distributed Systems and Networks," in *OSDI*, Boston, MA, Dec. 2002, pp. 255–270.

[14] Y. Ahmad, U. Çetintemel, J. Jannotti, A. Zgolinski, and S. B. Zdonik, "Network Awareness in Internet-Scale Stream Processing," *IEEE Data Eng. Bull.*, vol. 28, no. 1, pp. 63–69, 2005.

[15] P. Pietzuch, J. Ledlie, J. Shneidman, M. Roussopoulos, M. Welsh, and M. Seltzer, "Network-Aware Operator Placement for Stream-Processing Systems," in *ICDE*, 2006.

[16] M. Akdere, U. Çetintemel, and N. Tatbul, "Plan-based Complex Event Detection across Distributed Sources," *VLDB*, vol. 1, no. 1, 2008.

[17] G. T. Lakshmanan and R. E. Strom, "Biologically-inspired Distributed Middleware Management for Stream Processing Systems," in *Middleware*, 2008.

[18] Y. Zhou, B. C. Ooi, K.-L. Tan, and J. Wu, "Efficient Dyanmic Operator Placement in a Locally Distributed Continuous Query System," in *OTM Conferences*, 2006.

[19] N. Tatbul, U. Çetintemel, S. Zdonik, M. Chemiack, and M. Stonebraker, "Load Shedding in a Data Stream Manager," in *VLDB'03*, 2003.

[20] N. Tatbul, U. Çetintemel, and S. Zdonik, "Staying FIT: Efficient Load Shedding Techniques for Distributed Stream Processing," in *VLDB'07*, 2007, pp. 159–170.

[21] H. Feng, Z. Liu, C. H. Xia, and L. Zhang, "Load Shedding and Distributed Resource Control of Stream Processing Networks," *Perform. Eval.*, vol. 64, no. 9-12, pp. 1102–1120, 2007.

[22] T. Repantis, X. Gu, and V. Kalogeraki, "Synergy: Sharing-aware Component Composition for Distributed Stream Processing Systems," in *Middleware*, 2006.

[23] Y. Zhou, K. Aberer, and K.-L. Tan, "Toward Massive Query Optimization in Large-scale Distributed Stream Systems," in *Middleware*, 2008.

[24] O. Papaemmanouil, U. Çetintemel, and J. Jannotti, "Supporting Generic Cost Models for Wide-Area Stream Processing," in *ICDE*, 2009.

[25] S. Seshadri, V. Kumar, and B. F. Cooper, "Optimizing Multiple Queries in Distributed Data Stream Systems," in *NetDB*, 2006.

APPENDIX
COMPLEXITY PROOF

Theorem A.1: The query planning problem (III.8) is strongly $\mathcal{NP}$-hard.

Proof: Consider an instance of the bin packing problem. Thus, we are given a number of B bins, a bin size V_0 and a list of items $\{1, \ldots, n\}$ with sizes $V_1, \ldots, V_n$. The objective of the bin packing problem is to find an assignment of the items to the bins that respects the size limitations of the bins. Formally speaking, we seek a partition $J_1 \cup \cdots \cup J_B$ of $\{1, \ldots, n\}$ into B subsets such that $\sum_{i \in J_h} V_i \leq V_0$ for each $h = 1, \ldots, B$.

We construct now an instance of the query planning problem based on the data of the bin packing instance. This query planning instance has one base stream, n query operators, n composite streams and B hosts. Each query operator has the single base stream as its input and a unique composite stream as its output. We set $\gamma_o = V_o$ for each $o \in \mathcal{O}$, that is, the amount of computational resources consumed by operator o is identified with the item size V_o. We also set $\zeta_h = V_0$ and $\beta_h = \kappa_{hm} = +\infty$ for all $h, m \in \mathcal{H}$, that is, we consider a system in which each host has the same computational resources given by the bin size V_0, while there are no bandwith constraints. The binary variable δ_s is set to 0 for the base stream and to 1 for each of the composite streams meaning that all composite streams are requested. Finally, we set $\lambda_1 = 1$ and $\lambda_2 = \lambda_3 = 0$. This implies that we aim to maximise the number of satisfied queries. Note that the best we can hope for is to satisfy all queries, in which case the optimal value of the query planning problem is given by n.

If the bin packing problem is feasible, then we can use its solution $J_1, \ldots, J_B$ to construct a solution for the query planning problem whose objective value is n. In fact, since there are no bandwidth constraints, we can route the single base stream to any host h, where we can execute all operators $o \in J_h$ and thereby satisfy all queries for the corresponding output streams. As this is possible for each host in the system and since $\cup_{h=1}^{B} J_h = \{1, \ldots, n\}$, we can indeed satisfy all n queries. Conversely, if n is the optimal value of the query planning problem, then we can use its optimal solution to construct a feasible solution for the bin packing problem. Indeed, all operators executed at host h form a subset J_h of $\{1, \ldots, n\}$ with the property that $\sum_{i \in J_h} V_i \leq V_0$. Since all n queries are satisfied, all operators are executed at least at one host in the system. Therefore, the subsets J_h, $h = 1, \ldots, B$, constitute indeed a partition of $\{1, \ldots, n\}$.

The above argument shows that the bin packing problem is feasible if and only if the optimal value of the query planning problem is n. Thus, the query planning problem inherits strong $\mathcal{NP}$-hardness from the bin packing problem. ■

Corollary A.2: The query planning problem (III.8) remains weakly $\mathcal{NP}$-hard if there is only one host.

Proof: If there is only one host, the query planning problem can be reduced to the weakly $\mathcal{NP}$-hard knapsack problem (which constitutes a bin packing problem with only one bin). ■